An experiment platform for evaluation and performance testing of a membership protocol for FlexRay

Ronny Almgren, Andreas Lindström

December 29, 2007
Master's Thesis in Computing Science, 2*20 credits
Supervisor at CS-UmU: Mikael Rännar
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

The car industry is constantly trying to develop cheaper and safer cars. Most cars today relies upon mechanical systems for braking, steering, acceleration etc. If all mechanical systems were to be replaced by electrical systems the cars could weigh much less, and be cheaper to produce. Replacing a well tested and working system with a completely new could however be at the expense of safety. The new systems must be at least as safe as the old, preferably even safer.

This thesis describes the implementation and testing of a membership protocol. A membership protocol is a service intended for computer systems in vehicles. The computer systems consists of several processing nodes. Each node is capable of performing several tasks. A task is referred to as a process.

The membership service's job is to give each node in the system a view of how all the processes in the system are doing. This allow the system to compensate for a process that has failed. Allowing the system to tolerate faults is a necessity since it is impossible to create a completely fail safe system.

The membership protocol is implemented on a cluster with experimental nodes using FlexRay as communication controller.

The results shows that the membership protocol works very well and have good performance.

En experimentplattform för utvärdering och prestandatestning av ett protokoll för membership på FlexRay

Sammanfattning

Bilindustrin försöker ständigt utveckla billigare och säkrare bilar. De flesta bilar idag förlitar sig på mekaniska system för bromsar, styrning, gas osv. Om alla dessa mekaniska system skulle bli ersatta av elektriska system skulle bilarna kunna väga mycket mindre och dessutom bli billigare att tillverka. Att ersätta välbeprövade system med nya och otestade skulle dessvärre kunna bli på bekostnad av säkerheten. De nya systemen måste vara minst lika säkra som de gamla, helst ännu säkrare.

Detta examensarbete beskriver implementationen och testandet av ett membership-protokoll. Ett membership-protokoll är en tjänst tänkt att användas i datorsystemet i fordon. Datorsystemet består av flera noder, där varje nod kan sköta flera uppgifter. Varje uppgift kallas en process.

Membership-tjänstens uppgift är att ge varje nod i datorsystemet en bild över hur alla processerna i datorsystemet mår. Detta ger systemet möjlighet att kompensera utifall att en process har kraschat. Att tillåta att ett datorsystemet får fel är viktigt då det är omöjligt att göra det helt oförstörbart.

Membership-protokollet är implementerat på ett kluster med experimentella noder som använder FlexRay som kommunikationssystem.

Resultaten visar på att membership-protokollet fungerar bra och ger god prestanda.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The car industry constantly evolves, and the next step in this evolution is to computarize the systems in a vehicle that has previously been mechanical. This is however not something that is done with the flip of a hand. The major problem with doing this concerns safety. A sudden change in how vehicles operate could easily make them much more prone to losing its navigation capabilities. That is why there is a great need to do more research on this subject, which is the subject for this master's thesis project.

1.1 SP Technical Research Institute of Sweden

SP (SP Technical Research Institute of Sweden) is a research institute owned by the Swedish government. Its main geographical location is Borås, where there are about 850 employees working. There are also minor divisions placed in Stockholm and Skellefteå. Borås is where we have been stationed while working on this thesis.

SP's main tasks are research, technical examinations, measuring, quality assuring and certifications. SP is also obliged to calibrate equipment, such as gasoline pumps and scales for various uses.

The section of SP where this thesis has been performed, ELp (Electronics and Software), is the section for research on reliable and safety critical software. They are also certifying developers for using different standards. There are currently about 10 employees working at ELp.

SP is involved in the CEDES (Cost Efficient Dependable Electronics Systems) project, which is a project that researches cost efficient, intelligent safety systems. Among the other collaborators included in the CEDES project, Volvo and Chalmers can be found. Carl Bergenhem, our supervisor at SP, is working in this project with a membership protocol which will make the computer systems in a car safer. This is the subject of this thesis. The goal is to test the work Bergenhem has done in a real environment to see how it works and behaves outside a simulator. We hope to have contributed to the CEDES project by doing this thesis.

A previous project called GAST (General Application Development Boards for Safety Critical Time Triggered Systems), which had the goal to develop an experimental platform for testing of communication and processing in vehicles. The project was closed in may 2006, and resulted in physical hardware that can be used for software testing. The hardware consists of a computing board coupled together with a communication board, and each of those pairs is called a node. Several nodes can be connected which

ultimately forms a cluster. This is the hardware we have been using while doing this thesis. More information on this hardware will come in Chapter 3. SP, together with Volvo, Saab and several other participants, were part of the GAST project.

We came in contact with SP about this thesis through the National Degree Project Pool, which is a web-site for advertisements on master's thesis projects. Our supervisor at SP, Carl Bergenhem, had placed an entry at this pool for an interesting project concerning membership. We contacted Bergenhem and after a little conversation and a meeting we came to the conclusion that we should do this project.

1.2 The automotive industry

Ever since the first car was invented it has become more complex for each new car model that has been designed. It was not very long ago since the entire electrical schema of a car could fit on a single paper. Today even a simple thing such as the functionality of the indoor lights are difficult to grasp, and require much planning.

There are lots of new functionalities added as they are invented, such as ABS (Anti-lock Braking System), active suspension and ESP (Electronic Stability Program). Eventually the car industry plans to add x-by-wire control to the car. X-by-wire control means that as few parts as possible are controlled mechanically and rather controlled by a computer interface. The name x-by-wire comes from the term fly-by-wire which has been developed for aircrafts.

All this new complexity requires that safety in the car has to be taken to a whole new level. Despite all changes of the design the safety in the future car has to be as safe or more preferably even safer than in the cars of today. The evolution must go forward, not backward.

The car industry is also interested in making the production cheaper. Since reproduction of software does not cost as much as the production of an ordinary mechanical solution, the software solution is the advancing field. Another goal is to make the car lighter with less heavy mechanical solutions, which hopefully x-by-wire will solve to some degree. Although right now it seems as though the cabling in an x-by-wire car weigh just as much as the mechanical parts it is intended to replace.

This new technology requires more bandwidth for the communication, according to [8]. The average bandwidth needed for the engine and the chassis control is estimated to reach 1500 kb/s in 2008 while it was 765 kb/s in 2004 and 122 kb/s in 1994. CAN, which is the most common protocol of today, may be replaced with a new protocol called FlexRay because of its advantage that it has a higher bandwidth and support both for event triggered (as CAN) and time triggered communication. Time triggered communication is needed in safety functions since a guarantee for having communication with other nodes is required.

1.3 Fault tolerance

It is close to impossible to create a system that never suffers from a failure. As such the goal is to create a system that is safe enough and able to tolerate faults. If a fault occurs, the system will need to do whatever is necessary to be able to recover from that fault.

An example of this would be a brake application. If, for some reason, the brake for one wheel of a car fails the other brakes needs to be aware of this as soon as possible.

They may then be able to compensate for the loss of one brake and stop the car almost as safely as with all brakes fully functioning.

Another way to tolerate faults is to use redundancy. It is possible to do many measures to be adequately sure that the information arrives where it should.

Redundancy can be implemented on many levels. Information, hardware and software redundancy are good examples of this.

Information redundancy means that extra information is sent to be sure that the receiver does not misinterpret that data, e.g. due to disturbance on the channel. This can be done by sending the same data several times, or by using parity bits or checksums.

Hardware redundancy is easily examplified with the computer system of an aircraft. Aircrafts often have three independent computers that all perform the same calculations. The final result from these calculations will be the result calculated by a majority of the computer systems. This means that it is possible to have one computer break down and still have a working system. This method is however very expensive and is therefore not used for personal vehicles.

Software redundancy means that additional software is running on the system that allow extra precautions if something goes wrong. An example of this is the membership system which is explained in this report. This is preferred since software replication is much cheaper than hardware replication.

1.4 Membership

To solve the safety issues that have risen by introducing a new system design, there has been much research done in the computer science field. One of these research areas is to achieve an efficient and reliable membership service.

Membership services have been researched since the late 1970s, and there has been several variants developed since then. What they all have in common is that they all intend to give a consistent view of the correctly working entities in a system.

A membership service is used in a distributed network of computers nodes, called a cluster. Each node in the cluster may perform several tasks, which are referred to as processes. See Figure 1.1 for a graphical explanation of processes. The membership service maintains a list with the status of all processes in the cluster. The list is based on the messages sent by all the processes in the cluster. Each node independently builds the list by listening to the messages sent and performing the membership algorithm, which is the main subject of this thesis. By not having a master node the system will be much more fault tolerant as there will be no single point of failure. Figure 1.2 shows how a membership service works in a cluster with four nodes.

The membership service is needed to solve certain problems that may occur in a cluster. A particularly difficult fault may rise if a node in the cluster loses its incoming data connection, but not its outgoing data connection. Without a membership service all nodes would think that the node with the faulty connection are working correctly, except the faulty node which would consider all the other nodes faulty. Thus the fault wouldn't get detected.

All previously developed membership services only gathered the the status of the nodes in the cluster. The important property of the membership service handled in this thesis is that it operates on a process level rather than on a node level. Since it is inevitable that each node in the future vehicle computer systems performs several tasks this is a crucial feature for the membership service.

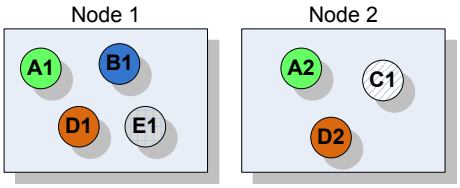

Figure 1.1: Nodes handles several tasks, each task is called a process. The membership service monitors all processes running on all nodes in a cluster. Processes typically form groups with processes on other nodes to perform distributed tasks. Process A1 is grouped with process A2 and process D1 is grouped with process D2 while the other processes (B1, C1 and E1) are ungrouped.

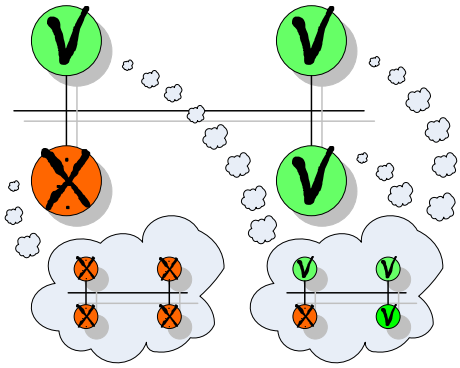

Figure 1.2: The agreed membership when one node has failed. The nodes are represented by the circles and they all are connected to a bus. The bubbles connected to the nodes shows the membership opinion of that particular node, a dysfunctional node got no valid opinion and hence considers all nodes to be dead. An 'X' in a bubble symbolize a dead or malfunctioning node, and a 'V' inside a bubble symbolize a working node.

A membership service can be implemented at either the hardware or software level of a node. Both have their benefits and problems. A hardware implementation got the advantage of being faster than an equivalent software implementation, but due to the fast CPUs of today and tomorrow this may not be such a big advantage as it used to be. The big advantage of a software implementation is that is capable of handling processes instead of only nodes.

The membership service that will be studied and implemented for this thesis is developed by Bergenhem and Karlsson [9]. This membership service is still under development, and is presented using an iterative approach where there are several versions, each becoming a little more advanced than the previous, so that the reader more easily can understand how it works. It will be implemented in software for a GAST cluster with FlexRay used as communication device. The version of the membership service that will be the final implementation is a version that handles multiple processes per node, and where the new membership list is only calculated when there has been a change in the system.

1.5 This report

The purpose of this document is to report the progress and conclusions of the implementation of a membership service.

To give the reader an understanding we will first describe the hardware and algorithms used to do the implementation.

In Chapter 2 we will describe the purpose and goals of the thesis. We will also describe the methods used to reach the goal.

Chapter 3 describes the hardware that was used to test the implementation.

In Chapter 4 we will describe the software environment that was used to do the implementation.

Chapter 5 will give a more thorough explanation of the communication system used, and will explain why it is the preferred communication controller.

Chapter 6 will explain what a membership service really is. It will also show on advantages of the membership service implemented in this thesis compared to other already existing services.

In Chapter 7 we will show how the implementation was carried through.

The results will be shown in Chapter 8 and the conclusions drawn can be seen in Chapter 9.

Chapter 2

Problem Description

As vehicles get more complex and new technologies gets adopted a great amount of research is needed to verify that the new technologies are at least as safe as the previous systems. New algorithms will be a necessity in these new systems, and it must be proven that these new algorithms work as specified.

A protocol for a membership service has been proposed by Bergenhem. The task for this thesis is to test this algorithm on real hardware to see if there are any unexpected complications. The algorithm has previously been tested in a simulated environment, but has never been tested on a physical environment with FlexRay. The thesis will not include proving that the algorithm is correct, it will be to verify the functionality of it by testing a limited set of black box test cases.

It will also be needed to measure the performance of the algorithm, in the implementation done for FlexRay.

There are several competing technologies for real-time communication that could be used in vehicles in the future, and hence it will be needed to decide whether FlexRay is the ideal candidate for this, and if this approach to a membership service is the best way to go.

The last task will be to do a demonstration of the implementation. As the test runs might be quite abstract, one may need quite a bit of insight in the implementation mechanisms to see whether it works or not, so a demonstration will be necessary to be able to show the uninitiated that it works.

2.1 Goals

The main objective for this thesis was very clear already from the start; to implement and test a membership protocol. In addition to the main objective few minor objectives were also defined.

The goal setup for this project was:

- Implement the membership protocol with dynamic reintegration for multiple processes as proposed by Bergenhem [9]. The implementation was to be done using an iterative approach.
- Test the performance and correctness of the implementation.
- Add support for dual channel usage in the implementation.

– Create a demonstration application to show that the membership protocol works as it should.

2.2 Purpose

The purpose of this project was to show that the membership protocol was correct and showed off good performance outside a virtual environment on realistic hardware. The membership implementation would also be a good platform to show an audience without knowledge about membership services how it is intended to work without going in too deep on the protocols characteristics.

This work would also result in a platform for testing and developing so called user programs. The platform can later on be extended to allow more uses in the future.

2.3 Methods

The tools that will be used for this project are the tools that is part of GEE (GAST Eclipse Environment) for development of the node application. The programming language used to write the node application will be ANSI-C. All the drivers and configuration tools are available, so there will be no need to write any assembler code for the node application.

To develop the test environment Java 1.6 will be used, all the Java source code will be written in Eclipse.

To allow both members of the project to work on the same code simultaneously, Subversion, which is a version control system, will be used.

The node application will be written using an iterative approach. The development cycle will follow the membership algorithms iterations, but will take an additional iteration to be able to handle multiple processes, which is not handled in the final version of the algorithm.

The tools needed will be updated as the project moves along, as it is hard to predict all the features we may want to have for the final test platform.

2.4 Related Work

There are a number of theses written that has handled the same subject as this thesis. The most related thesis has been performed by Bergström and Högberg [10], which also is the thesis that we have continued upon in this project.

The thesis project done by Vorkapic and Myhrman [18] has resulted in the drivers for FlexRay which we have used, and also provided us with an easy to use FlexRay configurator.

Chapter 3

Hardware

The hardware used in this project derives from a project called GAST (General Application Development Boards for Safety Critical Time-Triggered Systems). To develop the software and supply the nodes with runnable binaries ordinary PC:s are used. To simplify the communication between a GAST cluster and a PC, a PC with several RS232-ports are used for communication.

3.1 GAST hardware

The boards from the GAST project support four communication controllers that are interesting for the car-industry. The controllers are CAN, FlexRay, TTCAN and TTP/C. The GAST project has also developed two different types of ECU (Electronic Control Unit) boards (G1 and G2) to run simulations on. All the GAST hardware are of Open Source hardware design and has basic software drivers for research and development.

The hardware parts from the GAST project used in this project are G2 boards (see Figure 3.2) and FlexRay boards (see Figure 3.3).

In our cluster one G2 board and one FlexRay board connected to each other via a passive backplane forms one node, our cluster consists of five nodes. All nodes are connected via FlexRay. The physical layer used by the FlexRay boards are RS485.

3.1.1 G2

The G2 board consists of two CPUs, one MPC565 and one MC9S12DG256B[1] Freescale microcontroller [12]. These two microcontrollers can control each other, the MPC can force the HCS12 into RESET and the HCS12 can give the MPC565 a non-maskable interrupt.

The communication between the PC and the G2 board is done over RS232. The software is uploaded over RS232 to the MPC565. It is also possible to do some debugging over the RS232 connection, by using the built in debugger. The software is compiled with a version of GCC that has support for compiling to the MPC565 chip.

The debugger on the G2 board is called G2DBG. It serves not only purpose as a debugger, it can also function as a bootloader on the board. The final version of this debugger has some limitations, one of those is that it handles all the exceptions in the

[1] Further on referred to as HCS12.

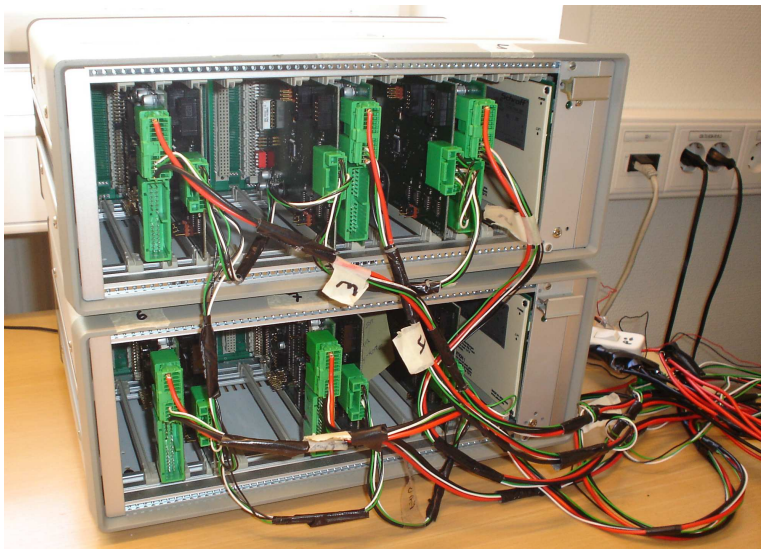

Figure 3.1: The GAST cluster used to test the node application. This cluster has five working nodes connected to each other as well as connected to a PC via RS232.

chip, and none of those are forwarded to the node application running on the chip. This limits the possibilities to make an application that may work as an operating system on the board.

A picture of a G2 board can be seen in Figure 3.2.

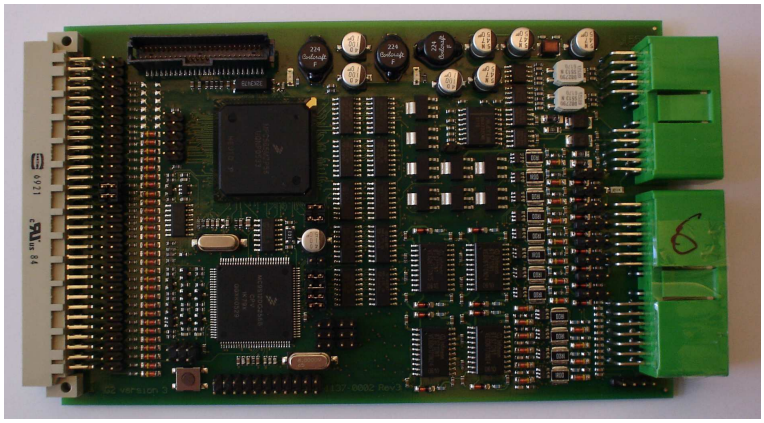

Figure 3.2: A GAST G2 board. The boards support four, for the car-industry interesting communication controllers, CAN, FlexRay, TTCAN and TTP/C. The GAST project has also developed two different types of ECU-boards (G1 and G2) to run simulations on. All the GAST hardware are of Open Source hardware design and has basic software drivers for research and development.

3.1.2 FlexRay

The FlexRay board has a Freescale MFR4200 chip and double RS485 transceivers, one for channel A and one for channel B, in FlexRay. The G2 board and the FlexRay board communicates via a passive backplane. The G2 board needs a 12 VDC power supply to operate, and the FlexRay board receives its power supply from the G2 board through the backplane. It is possible to connect several FlexRay boards to one G2 board to get more than one link.

A picture of a FlexRay board can be seen in Figure 3.3.

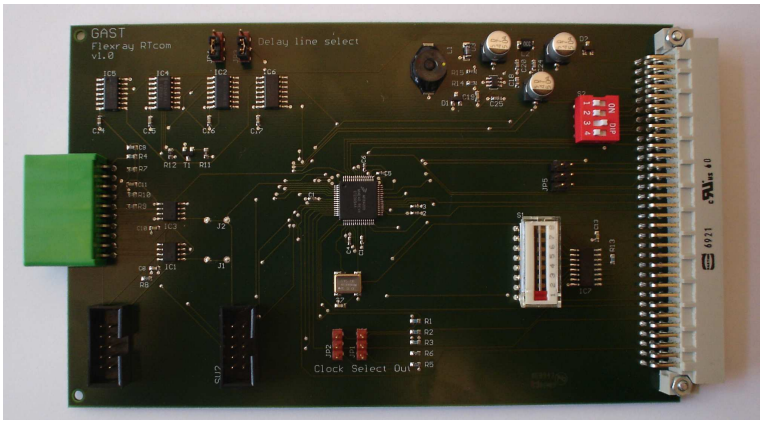

Figure 3.3: A GAST FlexRay RTcomm board.

3.1.3 Backplane

The backplane for a GAST node is of the passive type, which means that it does not contain any electronics. The purpose is just to make the connection of two, or more GAST boards easier. Further reading about for example what signals the pins correspond to can be done in the G2 Board Manual [13]. Figure 3.4 show what the backplane looks like.

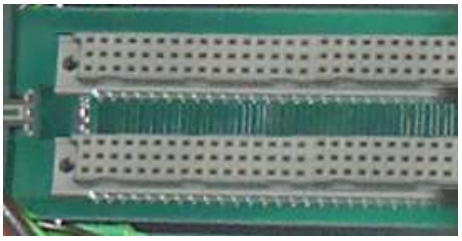

Figure 3.4: The backplane allows the G2 board and the FlexRay board to communicate with each other.

3.1.4 GAST BDM adapter

The GAST BDM adapter is a device used to update the software for the HCS12 controller on the G2 board. The BDM adapter is connected to the G2 board using a 6 pole flat cable, and then by a DSUB-9 female connector, using the standard RS232 protocol, to a PC. By using a specific piece of software called BDM12, the software for the HCS12 controller can be updated.

The BDM adapter was only used once throughout the project. A bug was found in the debugger/bootloader software for the HCS12 connector. The bug was not making the board unusable, but it was best to fix it so no problem would occur in the future.

The BDM adapter is shown in Figure 3.5.

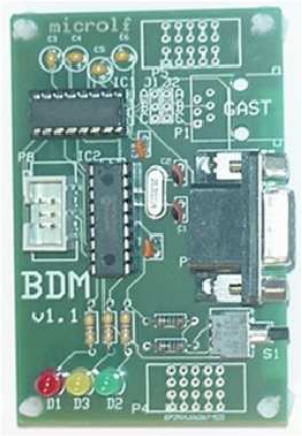

Figure 3.5: The GAST BDM hardware adapter used to update the software for the HCS12 controller.

3.2 Hardware setup

The first step to get this project running was to install the GAST cluster. As previous thesis workers had already used the same cluster it was assumed that it was as good as ready. Since it responded as it should have it appeared that it was working fine. However after more than a week of testing on the cluster with no success, there were suspicions that there was something wrong with most of the nodes in the cluster.

At that time it was uncertain whether it was the software or the hardware that was malfunctioning. A simple test program was received (that had been tested on another GAST cluster and was working on that) from Svenningsson and Chaudry who wrote [16]. The program was a simple 'Hello World'-application. After trying that program on the cluster it was obvious that the hardware was broken or had a bad setup.

The following week a great amount of error searching on the G2 and FlexRay boards was performed. A few suspicious signals was found, e.g. a disable signal from the G2 board going to the FlexRay board showed a high-frequency wave signal, while it should have been a constant high signal to not disable the communication controller.

After much work on this problem but no solution found a meeting was arranged with an expert on the subject, Roger Johansson, whom has written [12]. A few tests on

the GAST cluster was done by Johansson, and he concluded that most of the FlexRay boards were broken. The broken boards was replaced and the GAST cluster was finally working.

Chapter 4

Software

To develop software for the GAST nodes it is of great convenience to have the right tools to ease the process. Many tools were already available, and those will be introduced in this chapter.

4.1 GAST Eclipse Environment

GEE (GAST Eclipse Environment) is a package containing several tools used to develop software for the GAST hardware. It contains Eclipse, GAST plug-in for Eclipse, Cygwin and a GCC compiler with support for the MPC565 architecture. In this thesis GEE has been used to develop all the software for the GAST cluster. The following sections will describe the tools included in GEE more thoroughly.

4.1.1 Eclipse

Eclipse [2] is an IDE (Integrated Development Environment) written in Java. It is primarily designed for writing Java code, but has plug-ins to support C, C++ and several other programming languages.

GEE contains a plug-in to upload files to GAST nodes via Eclipse. This plug-in was however very slow and inconvenient to use, thus it was decided not to use this plug-in.

4.1.2 Cygwin

Cygwin [1] is a Linux like environment for Windows, it provides all the most necessary Linux commands in Windows. This made some tedious work much easier by giving a more powerful environment for creating scripts to automate repetitive tasks and creating Make files.

4.1.3 GCC

GCC (GNU Compiler Collection) [4] is the compiler used to create the executable binaries for the MPC565 processor on the G2 board. GCC has been used together with Make files to create the files usable on the G2 board.

GCC creates S19 files from the ANSI-C source code. S19 files are the binary file format accepted by the MPC565 architecture.

4.2 ODEEP FlexRay Configurator

A FlexRay board needs to be configured by the node application before it can be used. It is configured by entering a configuration state, while in that state all the necessary parameters are written by changing the internal registers on the FlexRay board. The registers are written via the backplane from the connected G2 board. There are quite a few parameters that needs to be set, and since each board will need a unique setup there is much that can go wrong if it was to be set up manually.

For these reasons it was a great advantage to use the available FlexRay Configurator. The FlexRay Configurator provides a graphical interface for the setup of the FlexRay board in the cluster. The application was written by Vorkapic and Myhrman [18] during their master's thesis project. While this application does have a few problems and limitations, it simplified the setup a great deal.

The configuration done by this application is exported to C files, which is used in the node application and contains all the setup needed.

A screenshot from the main interface of ODEEP FlexRay Configurator is shown in Figure 4.1.

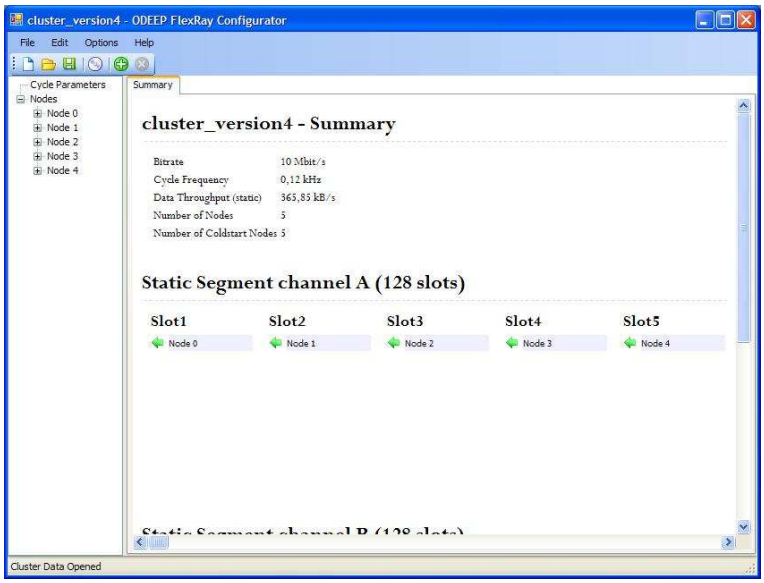

Figure 4.1: The main screen of the ODEEP FlexRay configurator. The main screen shows a time-line with each node's sending and receiving slots.

4.3 Subversion

SVN (Subversion) [6] is an open source version control system. It allows users to keep track of changes made over time in any type of electronic data. It is also used to allow several members of a project to work on the same code. If there are several members working on the same code it will merge the code, and if collisions appear it lets the user repair that code manually.

SVN is written with the intention to replace CVS (Concurrency Version System). SVN has grown quite popular, and has now surpassed CVS in Open Source projects.

Since Eclipse was used as the development environment it came in handy to use the plug-in to Eclipse called Subclipse [5]. This plug-in provided all the functionality given by Subversion integrated in the Eclipse environment.

Chapter 5

The FlexRay communication system

FlexRay is a communication system for advanced automotive control applications. It is the result of a cooperation between a couple of car manufacturers and some electronics manufacturers. To organize the work they started the FlexRay Consortium in September 2000, the original members were BMW, Daimler-Chrysler, Philips and Motorola. Since then the Consortium has grown to include some of the automotive industry's largest and most influential players, including Bosch, General Motors, and VW among others [3].

5.1 Description of FlexRay

FlexRay is a time-triggered architecture with support for event-triggered communication. FlexRay has support for dual channels which gives higher fault-tolerance and/or increased bandwidth. The maximum bandwidth in a FlexRay network is 10 Mbit/s. FlexRay supports two different kinds of cluster configuration, a bus network or a star network. These two network types can be combined or used separately over one or two channels [7].

The FlexRay communication protocol has all the benefits of a time-triggered communication protocol such as the built-in knowledge of when an alive node is supposed to transmit. In addition to this it also has support for a collision-free bus access and guaranteed message latency. There are also an independent bus guardian that provides further support for error containment.

A FlexRay cycle consists of one static part and one dynamic part (see Figure 5.1).

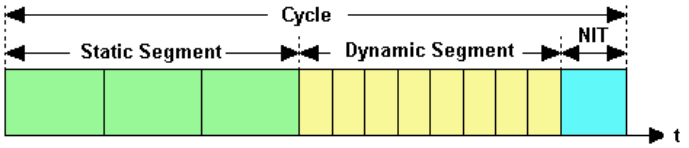

Figure 5.1: The FlexRay cycle starts with a static segment, followed by a dynamic segment and finish with a network idle time, NIT.

The static segment consist of a configurable number of slots, each slot containing room for the same amount of data. Every message is broadcasted to every other node in the network. If one or more nodes has a greater need for communication that node can be assigned to several slots which gives it a larger static piece of communication. It is also possible for a node to only listen to a channel by not being assigned to any slot at all. The slots that are not assigned to any node and the slots assigned to broken nodes will be silent.

The dynamic segment consists of mini-slots. Each mini-slot will be extended to a full-size dynamic slot when the node assigned to the current mini-slot has a message to transmit. Like the static segment each mini-slot is assigned to one node. The first slot is the one with the highest priority since it is the first to be able to expand to a full-size slot. It is not possible for all mini-slots to expand to a full-size slot, the result of this is that the dynamic messages with the lowest priority can suffer from starvation.

Network idle time, NIT, is the time in the end of each cycle. This phase is used for synchronization of the timing in the protocol. The time in a FlexRay cycle is measured in macro-ticks, mT. Each mT consists of number of micro-ticks, μT, which is the smallest time unit in FlexRay. One mT into the NIT segment a recalculation of how many μT there are in one mT starts. This is done every cycle, locally at each node.

The division between static messages and dynamic messages is configurable and depending on the network requirements it is possible to make them larger or smaller compared to each other.

A slot contains a frame that consists of a header segment, a payload segment and a trailer segment (see Figure 5.2). The header segment consist of information about the frame, such as the payload length, cycle count and different indicators. The header segment also has a field for a cyclic redundancy check (CRC) to keep track of that the header does not contains any errors. The field 'payload length' contains information about the size of the payload segment. The payload segment contains the data transmitted by the current node and the last segment, 'trailer segment' contains a CRC over the whole package.

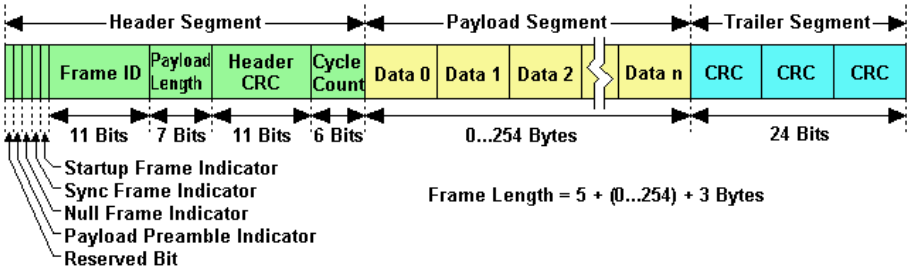

Figure 5.2: A FlexRay frame looks the same way if it is in the static or the dynamic segment.

As an extra precaution a node can only join one channel at a time. This is because a failing node must not be able to disturb the whole network. It is also possible to make a node a start up node or not. Only the start up nodes are allowed to try to initiate communication which reduces the chance of a failing node to disturb.

5.2 MFR4200

Freescale semiconductor manufacture and sells FlexRay communication controllers, one example is the MFR4200. The implementation in a MFR4200 chip follows the FlexRay protocol specification.

During normal operation the MFR4200 is in a state called 'Normal active Operation'. All states are represented by a number, 'normal operation' is number two. The MFR4200-chip stores this state in a buffer that can be read by the CPU connected to the MFR4200-chip.

The startup procedure starts when two or more modules are connected to each other and at least one of them is a startup node. When a node is powered up it enters a state called 'configuration' state. In this state a startup node may make a configurable number of attempts to start up the communication, if this fails the node will fall back into the 'integration listen' state. To make a new series of attempts the node needs to re-enter the 'configuration' state.

The communication of FlexRay is based on a time-triggered design, which means that the physical channel is divided into time segments for each node to transmit in. The timetable is defined before startup and distributed to all nodes. To make all nodes work together and know when a certain node is allowed to transmit all the nodes has to be synchronized.

To Synchronize messages in a higher layer the MFR4200 provides a cycle counter that counts the cycle up to 63. This counter can be read and compared to the counter in the header segment of each FlexRay frame (see Figure 5.2).

5.3 Comparison of FlexRay, CAN and TTP/C

A time-triggered communication protocol has the benefit that it is foreseeable when a certain node is allowed access to the channel. A disadvantage due to this is that if a node requires less bandwidth for a period of time the bandwidth assigned to that node go to waste, hence a event-based communication protocol can be preferred. In a event-based protocol the access to the channel is triggered by an event at a node and if channel is idle the message is sent right away. If the channel is not idle there are different ways to decide for how long to wait and whos turn it is to use the channel next. The decision can be made based on an taking turn or an priority algorithm all depending on what requirements there are on the protocol.

5.3.1 TTP/C

TTP/C (Time Triggered Protocol class C) is an example of a time-triggered communication protocol. TTP/C uses Time Division Multiple Access, TDMA, to access the physical channel. This means that the channel is divided into timeslots for each node to transmit in. TTP/C has a built in system for supervision of which node are members in the cluster at a certain time. The bus has, simular to FlexRay, a way of synchronising the opinion of the current time. TTP/C is a newly developed protocol with support for many different types of physical layer. The highest bandwidth that can be achieved is 25 Mbit/s.

5.3.2 CAN

Controller Area Network, CAN, is a event-triggered communication protocol. CAN was developed in 1980 by Robert Bosch GmbH for the automobile industry to connect the ECU in a vehicle. CAN uses a priority algorithm to decide which node to gain access to the channel first. After a message has been transmitted all nodes that want to transmit sends a unique binary code onto the channel. During the low signal every node listen for an other node, if there are an other node transmitting that node has a higher priority (see Figure 5.3) and the node with low signal terminate its transmission.

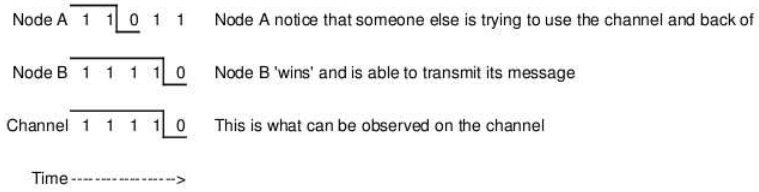

Figure 5.3: The node with the highest binary code wins since the most significant bit is sent first.

Depending on what type of physical layer that is used different bandwidth is achieved. The highest possible is 1 Mbit/s.

5.3.3 Requirements and comparison

In a safety critical real time distributed application the communication between all nodes is very important. It is also important to have sufficient bandwidth for all time critical messages not to be delayed through communication. The application communication over the network also needs to know if the node that the message is concerning is up and running, to be able to compensate for a malfunctioning node.

To make sure that a message will not be delayed more than a certain time a time-triggered communication like TTP/C can be used. Another solution is an event-triggered environment like CAN. This however can only be used if the requirement to be able to guarantee that a message should be delivered at a certain time can be ignored because CAN is a priority based communication protocol. This requirement can not ignored in a safety critical application.

The benefit of CAN over FlexRay is that it is widely used and the components are well known and mass produced. Because of the safety requirement and the prognosis that the need of bandwidth will increase [14], CAN is no longer sufficient (see Figure 5.4).

The use of only time-triggered such as TTP/C is an good alternative for safety critical applications but since FlexRay has support for both time-triggered and event-triggered communication car manufacturers may be more interested in a communication system that can be used for all different applications in a vehicle.

FlexRay does not have a membership service built in like TTP/C has. This means that if there are a requirement for such a service, that service has to be implemented in software instead of a hardware solution like TTP/C. The membership service in TTP/C only monitors nodes, but when implementing a service in FlexRay one can

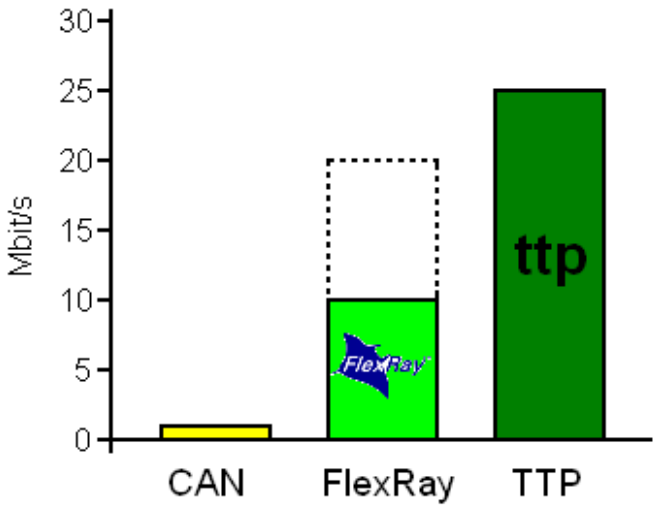

Figure 5.4: From left to right, CAN 1 Mbit/s, FlexRay 10 Mbit/s (possible 2*10 Mbit/s without the data redundancy) and TTP 25 Mbit/s.

choose an algorithm similar to the one presented in this thesis, written by Bergenhem, that supports monitoring of processes.

5.4 FlexRay configuration

A limitation for the posting of static messages is that the transmitted frame needs to be written down and committed in the transmit buffer one slot and 4 mT before the current slot that the transmission is supposed to be done in. That would be in the end of slot $n + 3$ in Figure 5.5.

Since G2 boards in the GAST environment handles all the exceptions without supplying them to the executable programs the solution has to poll for events in the communication.

It would be possible to have just two slots in between but that would require an interrupt when the data from slot n is available and directly after that write down the next message. With three slot in between it is possible to write down to the transmit buffer in the end of slot $n + 2$, in other words, merging slot $n + 2$ and slot $n + 3$.

The end of all slots is needed to poll for the next slot. To be sure of that no critical moment is missed the NIT and the in-between slots time was configured long, almost exaggerated. This made the use of the bandwidth very poor but this was not the main thing to test.

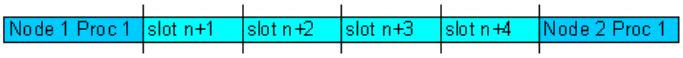

Figure 5.5: The messages transmitted from Node 1 Proc 1 will not be available for the other nodes in the cluster until sometime in slot $n+1$. However, since all the actual work is done in the beginning of each slot there is no guarantee that the message has arrived, therefore it is necessary to wait until slot $n+2$ to be sure of received that message. In this slot, $n+2$, it is possible to do calculations on the received data from slot n. After the calculation is done the next slot, slot $n+3$ is used for writing down to the transmit buffer.

5.5 FlexRay today

Today FlexRay is used in BMW X5 as a testplatform to gain experience with FlexRay. FlexRay handles the communication between the suspension in the car to make them work together to make a comfortable ride. It is not as safety critical as steer-by-wire application but it can result in strange driving characteristics if the communication fails. To use FlexRay in more safety critical applications, such as brake-by-wire, it needs to use both channels and perhaps two different power supplies. The batteries that are used today makes the benefit of weight saving disappear if the new system is compared to a braking system of today.

5.6 FlexRay in the future

FlexRay is a comparatively new communication system for the car industry. The prospect of success for FlexRay are good since there are several large car manufacturers that have invested time and money into this project. The FlexRay Consortium has also members from the electronics manufacturers which gives the project someone who's interests is to make a market for the hardware.

Chapter 6

Group membership services

A group membership service is an important abstraction for the next generation of communication systems. To support safety-critical applications for real-time environments it is necessary to be able to reliably tell which of the other entities in a cluster are working as they should, where each entity is either a node or, as in this case, a process running on a node.

Redundancy management is used to tolerate hardware failures during operation of the system. Hardware failures are nearly impossible to avoid, and thus it is important to design the system in such a way that it tolerates faults.

Grünsteidl [15] [11] defines redundancy management as consisting of three services:

- A timely and dependable communication service.
- A membership service that provides each component with consistent and timely information about the operational state of the other components.
- A reconfiguration service that allows to remove faulty components from the system and to re-integrate components into the system.

The first service is a prerequisite of any distributed real-time system. The other two are needed for redundancy management while being useful as a basis for other services, too. A redundancy management service should not degrade the real-time communication service provided by the distributed architecture. The redundancy management service must be fault tolerant and it must possess real-time capabilities. For a membership service to be usable in a real-time system, it has to fulfill two requirements; it must be consistent and timely. Consistency means that all correct nodes have the same membership information. Timely membership information refers to the property of the membership protocol that any state change of a communication node from active to inactive (or vice-versa) at time $T-\Delta$ is known to all other nodes at time T. Furthermore a node should be able to know about its own state at time T if it did not fail or shut down [15].

There are already many existing protocols for a membership service in use today. Most notable of those is the membership service included in TTP/C. The membership service in TTP/C is implemented in the communication controller, and maintains the membership agreement by regularly sending the membership information to each node in the system. Most membership protocols available today does in fact use this method. It is however obviously not the most efficient way to keep a consistent view across all

nodes. A common restriction in most available protocols, including TTP/C, is that they only allow a very strict amount of node failures in a given time division.

The protocol that is studied and implemented in this thesis is developed by Carl Bergenhem and Johan Karlsson [9]. Bergenhem and Karlsson's proposed protocol is in turn based on a protocol described by Velério Rosset et al in [17]. The protocol developed by Rosset et al. only have support for nodes. The major contribution from Bergenhem and Karlsson is the added support for processes which is a very important feature.

The proposed protocol is, in contrast to TTP/C, separate from the communication controller.

6.1 Enhancing safety in a distributed environment

There are many distributed services that are required in the computer system of a vehicle. Braking, stability control and collision mitigation are a few examples. All of these requires a consistent view of which nodes and processes in the system that are functional. This can obviously be implemented as a separate service in each of these applications. However, to keep the safety to a maximum it is much more preferred to implement a shared service, hence the membership service. All additional programming in an application to handle these things is unnecessary and may be faulty implemented.

As the communication systems used in vehicles today mostly facilitate event-triggered buses they are unable or very bad at detecting faulty nodes. Since it is not possible to know whether a package is lost or if the nodes didn't intend to send a package when using a pure event-triggered communication system, one may never find out if a node is broken. It is however possible to add a time-triggered layer on top of the existing, and achieve a time-triggered system from a previously event-triggered. This has been done for many existing event-triggered systems, such as TCP and CAN. A time-triggered protocol is needed to implement a reliable membership service.

6.2 Proposed algorithm

The algorithm proposed for the membership protocol by Bergenhem and Karlsson can be found in Appendix H.3. An explanation of when the different phases are performed is shown in Figure 6.1.

The algorithm is executed once each cycle. Although, which can be seen in the algorithm description, the membership decision phases will only be executed if a change has been detected on any correct node in the system. The algorithm can handle that up to half (not including) the nodes fails every 2 consecutive cycles.

Bergenhem and Karlsson's protocol has not been formally verified as this is written. So an advantage of the membership service available in the TTP/C protocol is that there is a documented proof that the protocol is working correctly.

6.2.1 The heartbeat

The heartbeat is a flag sent together with the static message that signifies correctness. The phase in which the static data is sent and received is called the FD-phase (Failure Detection-phase). Each process that wants to be part of the membership group must send the static message each cycle with the heartbeat flag set. The heartbeat flag

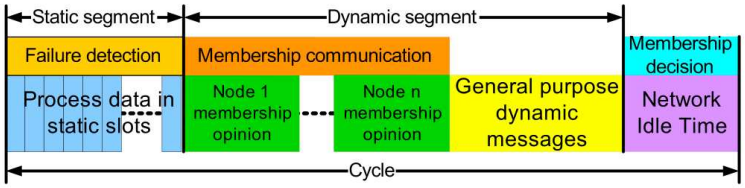

Figure 6.1: Illustration of a FlexRay cycle showing the segments and which membership phases that corresponds to the segments.

consists of one bit at the end of the message. The rest of the message contains all additional static data the process wish to send to other processes in the system, called the payload.

While receiving the static data a local opinion of which processes are working is calculated. The local opinion is a set, and may yet differ from the opinion on the other nodes.

$opinion'' = \{receivedheartbeats\}$

When all heartbeats have been received the opinion is updated again. It is intersected with the current membership view. This eliminates all the processes which may not be present in the current membership view. A specific procedure needs to be taken to join the membership.

$opinion' = opinion'' \cap membershipview$

In addition to the heartbeat flag there are two more flags in the static message. The second is the join flag. This flag is set when a process wants to join the membership view after having been excluded for some reason. The opinion is updated to include all the processes that wishes to join. The join flag is not available in the most basic version of the membership protocol, where it is not possible to have excluded nodes to join the membership.

$opinion = opinion' \cup joiningprocesses$

The third flag is the membership request flag. Any node that experiences a change in the system (e.g. a node that has suffered from a failure) also sets this flag in the static message. This will tell all nodes to perform the MC-phase (Membership Communication-phase) and the MD-phase (Membership Decision-phase). This flag is only used in the membership protocol with dynamic operation. *Opinion* is the candidate membership view. If it doesn't match the current membership view the membership request flag will be set.

Setting the membership request flag on a detected change will solve the problem with an ILF (incoming link failure, further explained in Chapter 7.1.3). As this will trigger the membership communication and decision phase, the decision function will detect that the node is faulty (the node won't get the membership communication messages and will therefore be excluded from the membership).

A node that joins the membership is not expected to have any knowledge about the current membership. It will assume the default values, that all nodes are alive. This view will not be counted as a vote during the membership decision phase. The real view will be found at the end of the cycle it sent its join message in.

6.2.2 Sharing opinion

In the very basic version of the algorithm the opinion calculated in the FD-phase is shared in every cycle. The phase in which the opinion is shared with all nodes is called the MC-phase. This provides an unnecessary amount of data to be sent every cycle. In the later versions of the protocol the MC- and MD-phase is only performed when a change has been recorded anywhere in the system. The news of a change in the system is transferred using the membership request flag.

When sending the opinion the dynamic segment is used. To be completely sure that the message is transmitted it is needed to give the message highest priority. If it doesn't have the highest priority the other dynamic messages might fill the dynamic segment, not allowing the opinion to be sent.

The message sent in the MC-phase consists of the opinion calculated in the FD-phase, the size of the number of members in the opinion and finally the gen-id (Generation-ID).

After the MC-phase has been performed the MD-phase (Membership Decision-phase) will be performed. This phase is just as the MC-phase only performed when a change in the system has been found.

The gen-id is a number which is incremented each time a change occurs. During the MC-phase all gen-ids received is stored in the GEN-ID-vector. The gen-id is used to exclude faulty members from the membership by removing all members which doesn't have $gen-id = max(GEN-ID)$. The gen-id will always be the highest for non-faulty nodes.

During the MD-phase strict majority voting will be used to determine the new membership view. The opinion that is calculated on more than half of the correctly working nodes will be the new membership view. Nodes have to be in the membership view for their vote to count. This prevents subsets of the system to create small isolated systems with their own membership.

6.2.3 Strict majority voting

The decision function's purpose is to let all nodes in the system have the same view. In the proposed algorithm the decision function is implemented with strict majority voting. This means that a candidate membership view needs to have at least half of the nodes voting for it. In case of a draw the result will be undefined, which leads to that node being excluded from the membership.

The algorithm for the decision function can be found in Appendix H.1.

6.2.4 Consensus

With the agreement calculated in the MD-phase consensus will always be reached within two cycles from when the fault was found, which easily can be shown informally. If the fault is found in the FD-phase by a correct node that has not yet sent its heartbeat consensus will be reached by the end of the same cycle.

As mentioned before this applies as long as the number of failures in two consecutive cycles are less than half of the currently correct nodes (which also are members of the current membership view).

6.3 Robustness

Much of this protocol relies on that nodes do not send any information when it has suffered from a failure. This is called the babbling idiot problem. A node that sends messages even while it has failed may erroneously be included in the membership when it shouldn't. This is solved by letting the node monitor itself, using a bus-guardian. A bus-guardian is a separate chip that monitors the communication controller and prevents it from behaving badly.

An aspect that may be important to look at is what a correct node that has lost its communication abilities with the rest of the system should do. While some systems should not do anything at all since they may affect the rest of the system very negatively, there are certainly less important systems that still may be able to do its job. This is however completely up to the developer of the processes, running on the nodes, to decide.

6.4 Minimizing communication overhead

The combination of the static/dynamic usage is what gives this protocol a very small overhead. The overhead during normal usage should be only 3 bits/process each cycle. This also makes this protocol a good choice even for larger systems where there are many processes. It scales well on a process level. The overhead when fault occurs in each cycle would be the size of the MC-message for each node. The size of the MC-message depends on the implementation, see Chapter 8.3.1 for a calculation of the overhead of the implementation done during this project. Since 64 nodes is the maximum possible due to restrictions in the FlexRay protocol, and much fewer nodes would be expected in a real system, this does not seem to be any problem.

6.5 Improving the protocol

There are many improvements that are possible to do in the membership protocol. Bergenhem is currently working on a protocol where the other nodes may see what type of fault that has occurred on a process or node.

The fault can be of such simple character as a CRC fault. A CRC fault could have been the cause of disturbances in the environment.

A CRC fault could be specifically handled. Since it is not unrealistic that such faults occur in some environments it would be a good idea to resend that message instead of excluding the process completely from the membership.

A threshold can be set on how many attempts of sending a process can do. Different processes should however have different values for the threshold. A brake process does of course need a smaller threshold than a system of window openers.

Chapter 7

Implementation

This chapter will describe the details of how the membership protocol was implemented. It will also describe the helper software and explain how and why the helper software was developed.

7.1 Node application

The node application is the software running on the GAST G2 nodes. The node application contains a rudimentary RTOS (Real-time Operating System), a membership service, and a few demonstration user applications. The node application is in the S19 file format, which is the Motorola PowerPC format of executable files. The debugger available on the G2 board is used to upload and run the node application.

7.1.1 Membership algorithm implementation

The development process of this thesis did not follow the original plans. The first intention was to continue on the previous thesis done by Bergström and Högberg [10]. After some studying of the code from that thesis it was apparent that the code was not complete. After reading their report yet another time it was also quite clear why they were not able to finish it. The same hardware problem we had ran in to had struck them at the end of their project, but wasn't solved.

After a couple of days of trying to fix their code with no success we took the decision to implement it our selves from the very beginning. The previous code was still very useful. We took much inspiration from earlier theses, which also contained useful code for the simple things such as math functions, input/output functions and ideas of how to control the operating system without the use of interrupts.

Just as the protocol was explained we implemented it using an iterative approach, starting from a simple version and working up to the final version. However, for practical reasons the very first iteration of the protocol was not implemented. The first iteration of the protocol do not differ much from the second iteration in aspect of an implementation, hence the second iteration protocol became the first iteration of the implementation.

A graphical overview of the features in each iteration is found in Figure 7.1.

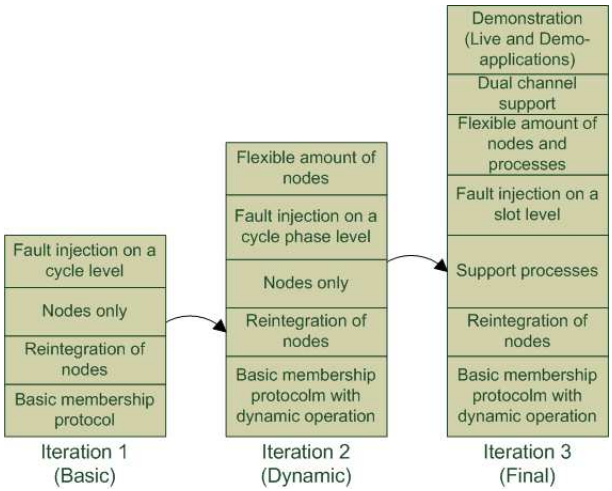

Figure 7.1: Illustration of the iterative development of the node application.

Membership Reintegration Version — first iteration

The first version of the membership implementation follows the second iteration of the algorithm, see Appendix H.2 for an algorithm description. This versions has support for the basic membership operations, including functionality to reintegrate excluded nodes.

This implementation only allowed for four nodes to participate. This restriction was caused by hard-coded data structures. As there was no possibility to use dynamic memory on the node all data structures had to be initiated when the program started.

The fault injection in this version was very basic. Fault injection is the action of artificially introducing faults into the nodes or processes, it is more thorougly described in Chapter 7.1.3. It only allowed faults to be injected on a cycle level. This made test runs of the membership algorithm correctness pointless, since it was not possible to test very many scenarios.

The implementation of this version gave us a great understanding of how to work with the GAST cluster, and this knowledge was of great importance in the later versions.

Membership Dynamic Version — second iteration

In the first version all the opinions of which nodes were alive was sent in every cycle. The dynamic version changed that so the necessary data was only sent when a change was detected. This reduced the bandwidth used by the membership protocol. The FlexRay configuration was updated to support this. The opinion data was sent using the dynamic segment of the FlexRay cycle, while 3 bits of the static data was used to send a heartbeat every cycle along with a detection bit which was set when a change in the cluster was detected.

Additional improvements on the implementation was also made. It was now possible to run the cluster with 3 or more nodes by only changing a constant variable in the code. Although, due to the development environment used the FlexRay configuration had to be updated if the number of nodes was changed.

In this version the fault injection was updated. It was now possible to inject faults

on a phase basis. The faults could be injected in either the failure detection phase or the membership communication phase. Since this still was not enough to be able to run the final tests, no conclusion on whether the algorithm worked correctly or not was drawn.

Membership Final Version — final iteration

The final version added several big changes to the implementation. The most important change in this version was the ability to specify multiple processes running on each node. Each process, called a user program, sends its own heartbeat together with the data it wishes to send. The membership decision phase is done on the node level, so this is handled by the underlying system.

Among other new features are the ability to use both FlexRay channels for communication. The implementation only used the extra channel for redundancy, i.e. the same data was sent on both channels. The receiving function then did a decision upon receiving data on which channel sent the correct message.

The fault injection was updated for this version. It could now handle fault injection down to a slot level. The test scenarios available were now possible to run on the implementation. With this change it was however more difficult to specify the fault for the node application. But because of the ScenarioParser this was no major problem when using the TestPlatform. The ScenarioParser and the TestPlatform are helper applications which are explained in Chapter 7.2.

Several more small features were added to the membership implementation. A user manual and all the features are available in Appendix B.

7.1.2 Application structure

The node application utilizes a layered structure. The most hardware dependent layer is the FlexRay drivers. These communicates directly with the FlexRay boards, but supply easy-to-use functions for sending and receiving messages.

The membership service is a layer between the user program and the FlexRay drivers. This gives a transparent membership service.

Wrapped around the membership service is a layer of fault injection

The RTOS controls the timing, and decides when user programs are allowed to execute its code.

See Figure 7.2 for a graphical diagram of the structure. Explanations of all parts in the node application is to be found later in this chapter.

7.1.3 Fault injection

To be able to control the test runs done on the GAST cluster it is somehow needed to be able to introduce faults into either the communication bus or to the hardware running the node application.

This can be done by introducing a disturbance node, or by modifying the software so that it supports fault injection. Since there was no special hardware for fault injection available for this thesis it was decided to be done in the software.

The fault injection is implemented by introducing a new layer on top of the functions that will be affected by the fault injection. The new layer provide dummy functions which in turn will call the original functions. The dummy functions will, depending on if a fault has been injected or not, block the function call.

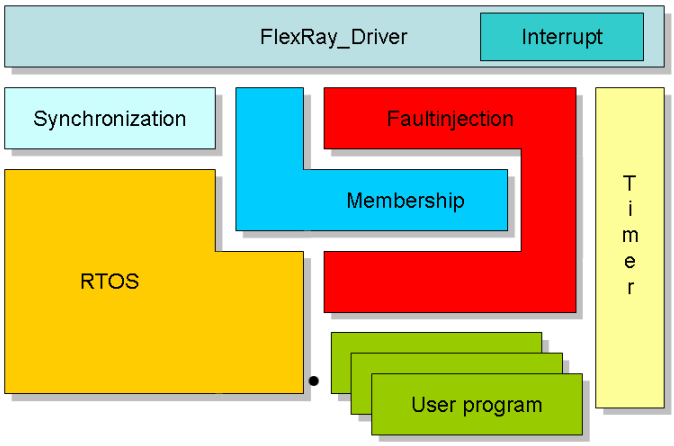

Figure 7.2: Layer diagram explaining how the node application is built up. The layer closest to the hardware is found on the top of the figure.

The faults can be injected on a node or process level. If a fault is injected on a node level it is possible to introduce faults to the communication for that node, by blocking the incoming and/or outgoing connection. A fault injected on a process level will only affect the communication for that process, by not allowing the process to send any messages.

Fault types

The events that are available on a node link level are listed in Table 7.1. These events simulate the status for a link (e.g. a cable connection) between two nodes. The link can be broken in both of the directions, resulting in either an incoming link failure (i.e. unable to receive data) an outgoing link failure (i.e. unable to send data) or both faults at the same time. An OK event tells the node that the link is all fine again and that data can be transmitted in both directions.

Event	Explanation
ILF	Incoming Link Failure, node can send but not receive.
OLF	Outgoing Link Failure, node can receive but not send.
OFF	Both Outgoing and Incoming Link Failure, node can not receive nor send.
OK	Node can both send and receive.

Table 7.1: Available events used for fault injection on a node level.

There are only two types of events available at a process level. A process can be simulated to have crashed. This means that there will be no heartbeat signal coming from that process. An OK event tells the system that it can allow the process to send heartbeats again. The events available on a process level is shown in Table 7.2.

Event	Explanation
OFF	The process will not send a heartbeat.
OK	The process is working correctly again.

Table 7.2: Available events used for fault injection on a process level.

Using these fault types it is possible to simulate practically any fault.

7.1.4 Synchronization

It is practically impossible to start the nodes at exactly the same time. There is always some random element disturbing, it may be in the connection to the power supply or different startup times for the chips. To solve this a synchronization function was added.

The synchronization function is not really needed to get the membership protocol up and running. Whenever three nodes in a cluster of five or six has been started the membership will get up since there will be a majority of nodes running. If all nodes are working correctly they will send the same opinion and calculate the same membership view.

The synchronization is used to be able to run the test scenarios properly. Without being able to synchronize it would be very difficult to detect when the test run is done. If the test scenario is specified to run for 20 cycles on a node, those 20 cycles could be over and done with when the other nodes has just started. The synchronization makes the nodes start at the same time, and after 20 cycles they will stop at exactly the same time.

For more experimental tests it is possible to disable the synchronization. The user may for example want to start the nodes at different times to see if the membership starts up correctly.

7.1.5 Timer

The FlexRay boards only have a clock that counts from 0 to 63 cycles. However, for some purposes this may not be enough. If the user would specify a test scenario where there are more than 63 cycles, the node application would get in to trouble. So a timer was added to the node application. This extends the FlexRay clock, and is able to count to the limit of what a 32-bit integer can store, making it possible to create very long test scenarios.

The timer also have another purpose. It acts as a fault detector that gives a warning if the node application would miss a cycle or enter a new cycle too late. It uses the internal FlexRay clock, which resets every cycle, to do this. The RTOS uses polling to determine when a new FlexRay cycle starts. After a new FlexRay cycle starts the user application get a small amount of time to perform its job. If this time is exceeded the node will be unsynchronized with the other nodes. If this happens the Timer will write a report of this so the node application can be modified to finish faster.

The RTOS can also, since it uses a polling approach, exceed the time limit if there is too much data to compute in a cycle. If this happens there are two ways to counter it. Either extend the FlexRay cycle, there is however a limit of how long it can be, or reduce the amount of work the chip has to do in a cycle. In an interrupt based implementation this would never occur. Any program that exceed its run time would be interrupted and the RTOS would handle the problem appropriately.

7.1.6 RTOS

The RTOS, or real-time operating system, in the node application is a simple as it ever could be. It consists of a while loop calling all the operations needed to keep the system running.

It handles non-blocking input and output, all the membership operations and runs each user program every cycle.

Even without any context-switching, memory control or other features an RTOS often is required to have, this RTOS was enough for the purpose of this application.

7.1.7 User program

A user program is a process running in the node application system. The simplest of all user programs would only send a heartbeat each cycle, and after that do nothing else.

The user programs can of course be more advanced. User programs can communicate with all other user programs running in the cluster. They use the static slot assigned to it to send any data.

The purpose of this thesis was not to create any specific user program, so only a few very simple programs were written to demonstrate the system, and they will be presented shortly.

The demonstration programs for the node application were intended to be very simple to implement. Two demonstration programs are available, and also one simple application that prints out the current membership on change. They all got in common that they are user programs that runs a very short piece of code each cycle.

The demonstration programs are called SampleProg, JageProg and MonitorProg, and they are all further described in Appendix B.5.

7.1.8 Interrupts

As mentioned before it was not possible to receive any interrupts at all using the debugger on the G2 boards. To get around this problem polling was used to continually check the interrupt registers which is set when an interrupt is thrown. The interrupts module was very closely tied to the FlexRay drivers and the RTOS, and in reality only contains two, very important, functions. To wait for a new slot and to wait for a new cycle.

This was however a very time consuming task for the processor, which will be shown in Chapter 8.3.

7.1.9 FlexRay configuration

The FlexRay configuration used in the implementation was built using the ODEEP FlexRay Configurator, which is described in Chapter 4.2.

The configuration makes use of both the static and the dynamic segment of a FlexRay cycle. The static segment consists of 128 slots, and the dynamic of 128 mini-slots. Since a mini-slot is much shorter than a normal slot the static segment makes most of the cycle.

To give the G2 board some additional time for calculations the NIT segment (i.e. the network idle time) is maximized. This extra time available is used to calculate the membership, and is also available for the user programs to run additional code.

Not all slots in the static segments are used for transmitting data. Throughout the project it was noticed that there seemed to not be enough time to send and receive data on the FlexRay bus. Some investigation revealed that the data that is to be sent need to be in the correct register at least two slots before it is really sent. The reason for this is some technical restrictions on the FlexRay board, where the controller need some time to lock the buffers, and copy the values to a shadow buffer. During the time the FlexRay board lock the buffers it's not possible to do any other computations. This is

a side effect of the FlexRay drivers, which intentionally tries to lock the buffer on the FlexRay board until it succeeds.

Even with two additional slots between each used slot there was not enough time to actually read the data on the previous slot, so two more slots were added between each used slot. So between each used slot there were actually four unused slots available only to let the G2 board do its calculations and gathering of data. The total amount of slots really used of the 128 slots specified in the configuration was 20. Since there were five nodes in the cluster each node had the possibility to run up to four processes.

It takes a relatively long time to access a register on the FlexRay board. Some measurements showed that it takes approximately one macro-tick (a slot consists of 55 macro-ticks in this FlexRay configuration) to read or write in a register on the FlexRay board from the G2 board. This is most likely due to a slow communication protocol between the two boards. However, since it takes several accesses on the FlexRay registers to read or write a message this really does take a great amount of CPU time on the G2 board. This would hopefully be solved in a commercial implementation of the node, where these two boards could be much more integrated with each other. Alternatively rewrite the drivers to make use of interrupts that only locks the buffer when it is possible.

A numeric analysis of the possible throughput using this FlexRay Configuration is shown in Chapter 8.3.1.

To reduce the amount of messages sent it is possible to specify a back-off-time. This is a value that determines how many cycles a process, not included in the membership view, will have to wait before it sends a join request. The back-off-time is reset every time a join request is sent. A back-off-time set to zero would send a join request each cycle until it gets included in the membership. Having an inappropriate back-off-time may, however, cause some problems which are discussed in Chapter 9.1.

7.2 Helper software development

The first method used to upload the node application binary files was to use the program HyperTerminal which comes as a default application with Microsoft Windows XP. The method required the user to write the command necessary to upload the binary file to all the nodes one by one. This was very time consuming, and an application that automated this was needed. The application, first called SimpleFileUploader later on renamed to SerialBatchServer, was developed throughout the project with more features added as they were needed. The application was written in Java and used the RXTX library for communication with the GAST cluster.

The very first version of this application only allowed a binary file to be uploaded to one node. Even though it still was that simple it was a big improvement from uploading via the HyperTerminal. This was still not enough, so later versions allowed binary file uploading to the entire cluster simultaneously. It was also added functionality for communication with the cluster from other computers via Ethernet.

For testing of the software a few input and output data interpreters were needed. The fault scenario file format we had decided to use was not optimal to be used for input to the node application. The node application had to be as small as possible to fit the memory and also to be as fast as possible to upload to the node, so any unnecessary parsing was not wanted. The parsing application for the input data took one scenario file and converted it in to several files, each file containing the specific input data for one node on the GAST cluster.

The output data interpreter was needed to save the amount of time needed to manually check the correctness of the result from the node application. Since each run of the node application in the GAST cluster resulted in no less than five output files, this data somehow needed to be compiled into a file that was easy for the human eye to interpret. The output data interpreter checks the validness of all files and summarizes it.

All of these applications were at last bound together with batch files, scripts that runs all the programs in the correct order.

See Figure 7.3 for a graphical representation of how these applications work together.

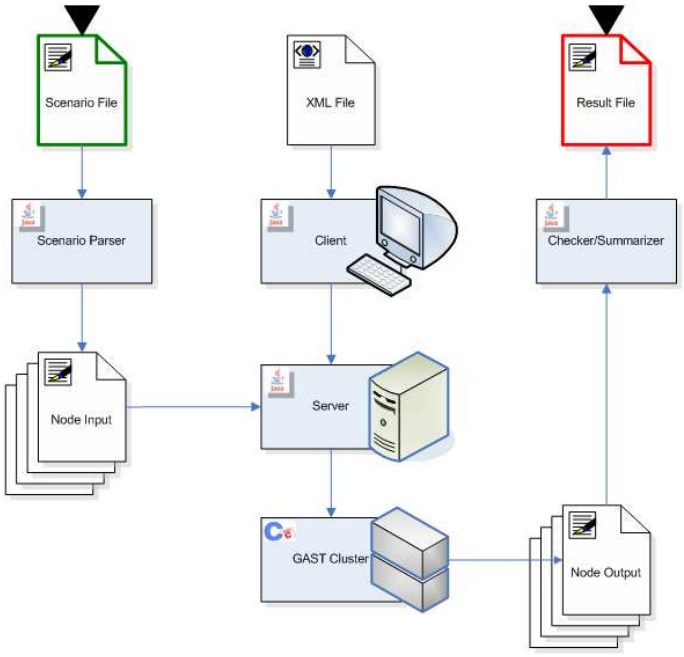

Figure 7.3: Graphical representation of how the implemented applications work together.

7.2.1 SerialBatchServer — automatic file uploading

The SerialBatchServer is an application developed for this project to ease the process of implementing the membership algorithm.

The server act as a front-end to the GAST cluster, and its purpose is to upload files and run the commands the users specify via an XML file, on each node in the cluster.

Users connect to the server front end via a client application, the client uses a TCP/IP socket to get a connection to the server. The client sends the commands that the users has specified in an XML file to the server.

It is possible to connect several users at once to the server. The server will queue the users up according to who connected first. After each user is done, the server will make sure to reboot the cluster using the automatic power switcher. This is needed since the program running on the node may get locked due to bugs in the code.

The server also allows another type of client to connect directly to a node. This makes it possible to try out things on the nodes more experimentally. This was very

useful when running the live demonstration on the node application, where we could pull the connection to a node and see all the other nodes correspond to that action.

For a more thorough explanation of what the server can do and how to use it, see Appendix E.

7.2.2 ScenarioParser — simplifying scenario specification

An application named ScenarioParser was made to simplify the submitting of a scenario specification. The ScenarioParser parses a text file and generates one unique text file for each node in the GAST cluster. The original text file contains both information that is the same for all nodes and also information that is specific for each node.

The choice of file format for the input data to the node application was not chosen for an arbitrary reason. Several tricky cases had already been expressed in that file format for a simulated version of the membership algorithm written by Bergenhem. These scenarios became the foundation for our testing. A few scenarios to test the simple scenarios was also written to test the core functionality of the node application.

Examples of parameters that are the same among all the nodes are the size of the cluster and the length of the test run. Parameters that are unique for each node is such as what type of fault and when it occurs. More about these and the rest of the parameters can be read in Appendix D.

7.2.3 Checker — summarizing the test run log files

The node application generates a log file for each node. These log files contains a selectable amount of data about the test scenario that was executed. The different types of data is, information about the membership vector, golden states (a parameter telling what the state is expected be if the application is working correctly) and so on, more can be read in Appendix C.

The Checker takes the data from all of the log files from each node and puts the interesting information together in one file. The Checker also do a comparison of the membership vector and the golden states to make a notification of that there is a difference. The amount of information in the summarized file is selectable, more about this can be read in Appendix C.

7.2.4 TestPlatform — tying it all together

The TestPlatform is a collection of batch-scripts used to tie the developed software together. The scripts are quite simple, and allow the user to run a whole set of scenarios with only one command. The output for all scenarios is stored in folders and the output will have to be reviewed manually by the user.

The scripts are `make.bat`, `run.bat`, `server.bat` and `terminal.bat`. See Appendix F for more information on these scripts.

Chapter 8

Results

The results from this project are gathered from a large number of test runs. The test runs ranged from longer tests to very short tests called scenarios. How the tests were performed and the results of the tests will be explained in this chapter.

8.1 Correctness verification

The main objective of this project was to verify that the membership algorithm worked correctly, implemented for a cluster consisting of GAST nodes. This was verified using a series of tests of the implementation. The tests had to cover as much realistic events as time permitted to be able to draw any final conclusions about the membership algorithm.

It is however important to remember that no matter the amount of tests done, it is impossible to show that the membership algorithm is completely fool proof using this approach. This is merely a verification that the membership algorithm works as it should for most cases on this specific piece of hardware.

8.1.1 Verification procedure

The tests, on which the results are primarily based upon, consists of a set of very short test scenarios. Each of those test scenarios is specified to be running for 20 FlexRay cycles, not all test scenarios facilitate all the 20 cycles though.

Even if 20 cycles does not sound much, it is more than enough to create a very big set of tricky test scenarios which should challenge the membership algorithm. The scenarios are specified in such a way that one or more nodes break down at a time where the algorithm is most likely not to be able to handle it.

The cycle frequency for the FlexRay configuration used in the implementation of the membership algorithm is 120 Hz. The test runs will therefore be completed very fast. Each cycle will be completed in 0,008 seconds, so each test run will take roughly 0,17 seconds to complete. With the overhead consisting of configuration, synchronization and data retrieving it will take a little more time to finish an execution of the algorithm, but that time is so small that it is of no value to count it.

To evaluate whether the implementation did finish correctly or not, a golden state is specified in the scenario file. If this golden state would not match the finishing state for the test run it is an indication that something has gone wrong during the test run.

The necessary test data is stored on each node in a matrix, and retrieved after the test run is done and stored as log files. The log files are interpreted with the Checker software (see Appendix C). Checker will make sure that the log file show correct data, and then write a new file with the summary of how good the test run has performed. The final step is to manually look at and verify the summary file.

8.1.2 Test scenarios

A test scenario is a test run of the node application. During that test run faults will be injected, which faults are injected are specified in the scenario. The faults are injected on a slot level in a FlexRay cycle, and as such it is possible to inject a fault wherever it is wanted.

The only faults possible to inject to nodes are faults on a link level, i.e. problems on the communication bus. With only link level faults it is however possible to simulate all the faults necessary, as a broken node can be thought of as a node with no communication to the other nodes at all.

It is also possible to inject fault to the processes running on the nodes. To simulate a process malfunctioning it is possible to switch the process off, and then back on again when the process is considered to be working again.

For the node application developed in this thesis project the file format used to specify the scenarios is a file format created by Bergenhem to initially be used for a membership simulator he created. In the scenario file there are a number of properties that must be specified, such as the total number of nodes and how many processes there are running on each node. Aside from all those properties the fault injections are specified. Those are specified by typing which node, which cycle, which slot and finally what fault will be injected.

Whenever a fault should cease it is needed to write a new event, which should be specified as OK. This event is specified in the same way as the other types of faults.

The test scenario file also contains a special type of event called gold. The gold event is available for testing purposes, and each gold event specifies how the membership vector should look at a certain point in time of the test scenario.

Totally there have been around 30 test scenarios run on the node application. The majority of the tests have contained the most tricky scenarios that could be thought of. But for testing the node application there have also been a few very simple test scenarios.

8.1.3 Correctness results

Using the TestPlatform all the test scenarios have been executed. The summary files of the test runs have afterwards been manually verified, and those results can be found in Table 8.2 and 8.3

See Table 7.1 and 7.2 for the an explanation of the available faults.

Table 8.1 shows an example of a simple test scenario. Figure 8.1 illustrates how the slots and phases corresponds to each other.

There were no tests to crash processes when testing on four nodes. However on the test results with five nodes a fault was injected on node level if the process column contains a dash ('-'). Otherwise it was injected on the local process with the number corresponding to the value in that column.

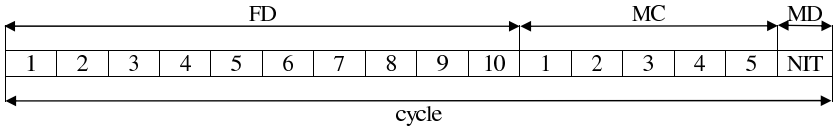

Figure 8.1: Membership cycle model that shows the different phases in the algorithm and where the processes sends the heartbeats in FD and the nodes sends the communication messages in MC. Slot 1 in FD belongs to process 0 in node 0, slot 6 belongs to process 1 in node 0, slot 2 belongs process 0 in node 1 and so on. Slot 1 in MC belongs to node 0, slot 2 to node 1 and so on.

Node	Cycle	Phase	Slot	Event	Result	Comment
1	2	FD	1	OFF	Success	
1	4	FD	1	OK		

Table 8.1: Example of a connection failure. This scenario is specified to introduce a fault on both the outgoing and the incoming connection for Node 1. The fault is injected in the second cycle of the test run, in the first slot in the failure detection phase of the algorithm. The connection is specified to become OK again two cycles later.

Node	Cycle	Phase	Slot	Event	Result	Comment
1	2	MC	1	ILF	Success	
1	4	MC	6	OK		
1	2	MC	1	ILF	Success	
3	3	FD	1	ILF		
1	4	MC	5	OK		
3	7	FD	1	OK		
1	2	FD	9	ILF	Success	
1	4	MC	5	OK		

Table 8.2: Test results for scenarios specified for four nodes.

Node	Process	Cycle	Phase	Slot	Event	Result	Comment
0	-	1	FD	1	OFF	Success	In this scenario there won't be a membership agreement.
1	-	1	FD	1	OFF		
2	-	1	FD	1	OFF		
3	-	1	FD	1	OFF		
4	-	1	FD	1	OFF		
0	-	2	FD	1	OK		
1	-	2	FD	1	OK		
0	-	2	FD	9	OLF	Success	
0	-	5	MC	6	OK		
0	-	2	MC	3	ILF	Success	
4	-	3	FD	0	OLF		
0	-	5	MC	6	OK		
4	-	5	MC	6	OK		
0	1	2	FD	0	OFF	Success	
0	1	4	FD	0	OK		
4	-	5	FD	1	ILF		
4	-	5	FD	2	OK		
0	1	2	FD	0	OFF	Success	
0	1	4	FD	0	OK		
1	-	5	FD	1	OLF		
1	-	5	FD	2	OK		
4	-	2	FD	9	OFF	Success	
4	-	4	MC	6	OK		
0	1	2	FD	9	OLF	Success	
0	1	4	MC	6	OK		
1	-	5	FD	0	OLF		
1	-	5	MC	6	OK		
2	-	5	FD	0	OLF		
2	-	5	MC	6	OK		
1	-	2	MC	3	ILF	Success	
1	-	2	MC	6	OK		
1	-	2	MC	3	ILF	Success	
1	-	4	MC	5	OK		
1	-	2	MC	3	ILF	Success	
1	-	4	MC	5	OK		
4	0	3	FD	0	OFF		
4	0	5	FD	0	OK		
4	-	7	FD	0	ILF		
4	-	7	MC	5	OK		
1	-	2	MC	3	ILF	Success	
1	-	4	MC	5	OK		
4	-	3	FD	0	OFF		
4	-	5	FD	0	OK		
1	-	2	MC	3	ILF	Success	
1	-	4	MC	5	OK		
1	-	2	FD	5	OLF	Success	
1	-	4	MC	5	OK		
1	0	2	FD	1	OFF	Success	

Node	Process	Cycle	Phase	Slot	Event	Result	Comment
1	0	4	MC	6	OK		
1	-	3	FD	3	ILF		
1	-	5	MC	6	OK		
1	0	2	FD	1	OFF		
1	0	3	MC	6	OK	Success	
1	-	3	FD	6	ILF		
1	-	3	MC	6	OK		
1	-	2	MC	0	OLF	Success	
1	-	4	FD	0	OK		
1	-	2	FD	0	ILF	Success	
1	-	3	MC	6	OK		
1	-	2	FD	0	OLF	Success	
1	-	3	MC	6	OK		
1	0	2	FD	0	OLF		
1	0	2	FD	5	OK	Success	
1	-	2	MC	4	ILF		
1	-	2	MC	6	OK		
1	-	2	MC	4	ILF	Success	
1	-	4	MC	6	OK		
0	-	1	FD	0	OFF		
1	-	1	FD	0	OFF		
2	-	1	FD	0	OFF		
3	-	1	FD	0	OFF		
4	-	1	FD	0	OFF		
0	-	1	MC	6	OK	Success	
1	-	2	MC	6	OK		
2	-	3	MC	6	OK		
3	-	4	MC	6	OK		
4	-	5	MC	6	OK		
1	-	2	FD	5	ILF	Success	
1	-	2	MC	5	OK		
2	0	3	FD	0	OFF		
3	0	3	FD	0	OFF	Success	
3	-	3	MC	1	ILF		
3	-	4	MC	0	OK		

Table 8.3: Test results for scenarios specified for five nodes.

As Table 8.2 and 8.3 reveal there has been no suspicous error found. The implementation of the algorithm works correct for all the test scenarios executed on the cluster.

8.2 Workflow

To do a test run of the implemented membership algorithm there are several ways to do it. One approach, with the advantage that it is quit easy to reproduce, is to make text files containing information about the GAST clusters structure and the fault scenarios to run.

To specify the fault scenario it is of course also possible to do it manually with the interface implemented on the GAST cluster with the connected PC or over TCP/IP via the connected PC.

The text file with information about the GAST cluster and the fault scenarios are parsed with a application written in java to generate a unique scenario file, one for each node. To use the text file together with the cluster, an XML file is required to handle the file system and the communication with the cluster.

When the test is performed the GAST cluster generates a output file from each node in the cluster. The output file contains the node's view at each cycle during the test run. Since all of this information is a little difficult to overview there is an application to merge the information from the different output files.

8.3 Performance

To measure the performance of the implemented membership algorithm one need to consider both the use of the CPU on the G2 board and bandwidth used for communication between the FlexRay boards. A third thing to consider is that the performance is dependent on the amount of fault occurring. If there are no faults at all, there is no need to do a voting among the processes. This saves both CPU usage and bandwidth. If there is a fault in a node the performance in this node may differ from the performance in other nodes.

For example if node 1 has an OLF (see Table 7.1), node 1 will try to reintegrate every cycle if the back-off-time is set to zero cycles, this will increase the amount of CPU usage. Since no other node in the cluster will receive the join request they will not need to do the voting and therefore have the same low amount of CPU and bandwidth usage until node 1 does not have an OLF anymore.

In case of an ILF in node 1 and back-off-time set to zero cycles, there will be a voting performed every cycle that votes against that node 1 should reintegrate.

8.3.1 Theoretical

To calculate the bandwidth used from one node in a cluster that is configured for our implementation there are several parameters needed to take into consideration. The first most obvious parameter is of course the FlexRay cycle frequence (FF). A second parameter is how much data in one cycle is dedicated to one node, this is depending on how many processes (N_PROC) are on the actual node. The rest of the parameters are how many bits used for overhead (OH), number of bytes (NB_s for static and NB_p for dynamic), size of a byte (BS).

These parameters inserted in the following formula will give how much data can be transfered during a second from one node when a certain per cent faults (fault%) occur, faults that leads to that a voting is necessary.

$$\frac{(NB_s * BS - OH) * N\_PROC + \frac{100-fault\%}{100}(NB_d * BS)}{BS} * FF = n(byte/s)$$

An example with a fault rate of 10% and the actual values from the implemented node application gives the following parameters:
$NB_s = 24$ bytes

$NB_p = 16$ bytes
$BS = 8$ bits
$OH = 3$ bits
$N\_PROC = 4$ processes
$fault\% = 10$ %
$FF = 0.12$ kHz

$$\frac{(24 * 8 - 3) * 4 + \frac{100-10}{100}(16 * 8)}{8} * 120 = 13068(byte/s)$$

The CPU usage can be calculated much more carefully and theoretical then just taking the time between the slot when all the data is collected and the slot when the result needs to be presented, but this at least gives a indication of the bandwidth usage.

8.3.2 Practical

As shown in 8.2 the CPU performance is very good. The difference between 8.2 and 8.3 depends on that the failing node tries to rejoin the membership every cycle, but the other node does not hear the attempt to rejoin in cycle four.

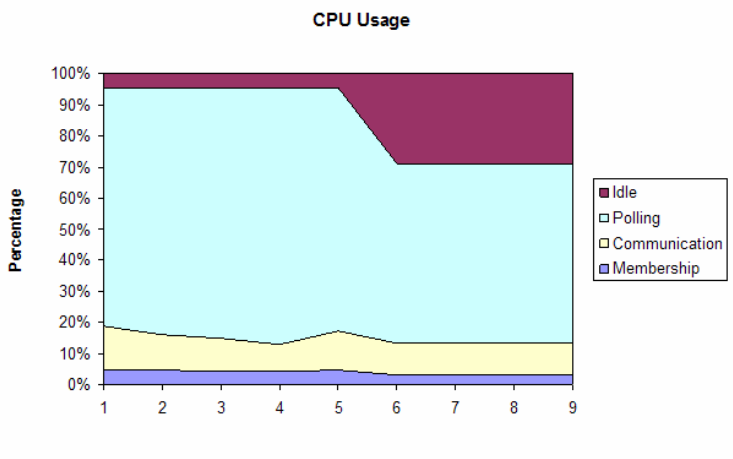

Figure 8.2: The CPU usage at cycle n in node 1 during a scenario where an OLF occurs in node 1 on cycle 4.

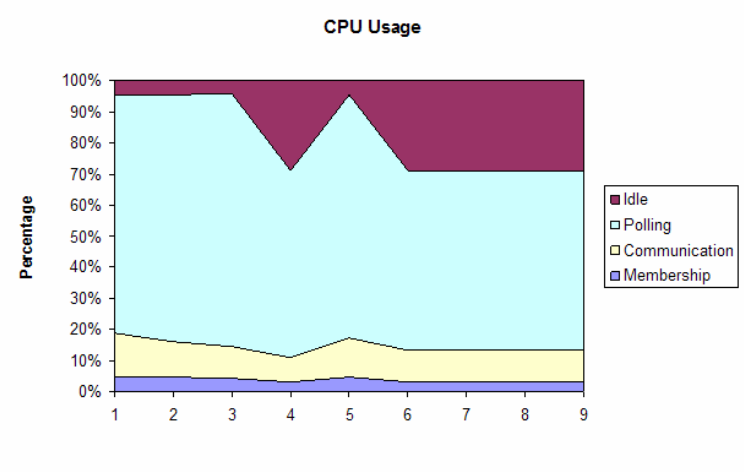

Figure 8.3: The CPU usage at cycle n in node 3 during a scenario where an OLF occurs in node 1 on cycle 4.

Chapter 9

Conclusions

This chapter contains our final conclusions drawn from the implementation of the protocol and from the results of the test runs on the implementation.

9.1 Conclusions from implementing the protocol

After doing this implementation a few things that one may need to consider has been discovered.

The first thing that comes to mind is that the FlexRay controller got some restrictions. Since it need the data available more than one slot before it is actually sent it is not possible to use the algorithm exactly as it is described. The algorithm suggests that if a node detects a fault in a slot, where the node is about to send a message, it should report this by setting the membership request flag in the message. Since this is not possible it has to be solved by either not setting that flag or by not allowing the membership group to have processes that sends their static message next to each other in the FlexRay configuration.

Storing the membership view in a set data type limits the expandability of the program. It is difficult to set additional parameters for each process in the set. If a new implementation was to be done we would recommend that it natively could handle parameters such as number of send attempts, what type of fault an excluded process suffered from etc.

One need to be careful with using the back-off-time. If the back-off-time has an inappropriate value the system may enter a deadlock where it never starts up. This may happen if for example a system with four nodes has a back-off-time set to two. If unlucky the first cycle there are two nodes trying to join (and start up) the membership. This will not be majority and they will not be able to. The next cycle the other two nodes will try to join with the same result. So it will continue, and the membership will never start up. Ideas to solve this could be to have a random back-off-time each join attempt. It could also be to set the back-off-time to a unique prime number for each node.

In general the protocol appears to be fairly generic, and will work with communication controllers that support either strictly time-triggered or time-triggered with a dynamic part such as FlexRay.

It is however by combining time-triggered and dynamic communication that the overhead of the protocol is kept so low. By using a strictly time-triggered communication

controller the opinion will need to be sent each cycle by all nodes in the cluster. Sending the opinion each cycle does obviously take more bandwidth than only sending it on changes in the system, but it still isn't that much of a problem. The size of an opinion depends very much on the implementation. In the implementation done in this thesis each process needs 1 bit in the opinion. This gives a fairly small size of the opinion. If each node sends the opinion each cycle it will still be a faily small overhead of the membership protocol, using a stricly time-triggered communication controller, making the protocol a good candidate also for such systems.

9.2 Conclusions from the results

The result data has showed that the membership protocol worked very well.

Performance measures reveals that the CPU usage of the algorithm is kept low. The bandwidth available when using the protocol is however very low compared to the maximum possible in the FlexRay protocol. Reasons for this include low performance drivers. It took quite long to send and receive messages for a node. The biggest limit for the bandwidth is however the FlexRay configuration used. The FlexRay configuration is designed to maximize the available CPU time, and does this by maximizing the void between each slot. This is in turn a side effect from not having a proper operating system available.

9.3 Limitations

One hardware restriction is about how far ahead the data has to be committed and the actual buffer unlocked before transmitting data. To transmit data in the static or the dynamic segment at slot n, the buffer must first be locked, the data must then be committed and the buffer unlocked not later than four micro-ticks before the beginning of slot $n - 1$ [7]. The existing drivers locks the buffers by repeatedly trying to lock it until it works. This obviously consumes much CPU time, and a new set of drivers which solves this issue would be needed to use this CPU time for something productive.

If the real-time operating system were to receive the interrupts sent from the FlexRay board it would be possible to only execute the data transmission only when it the FlexRay board is ready for it. This would alleviate the G2 board of some unnecessary work, enabling it to do perform more important calculations. This is most likely not a very difficult task, but would require the FlexRay drivers to be fairly closely connected with the operating system.

The ODEEP FlexRay configurator (see Chapter 4.2) limits the amount of slots available to use. To be able to utilize the maximum amount of slots in a FlexRay cycle it is needed to reuse buffers on the controller, this because there are fewer buffers than the maximum number of slots in the MFR4200 controller. When using the configurator each available buffer was tied to a specific slot, and the configurator provided no way to make it possible to change this during runtime. It would be quite difficult to solve this in the existing version of the configurator. Changing the buffers during runtime requires some code to be executed frequently to change the buffers when needed. This would most likely need to be closely tied together with the real-time operating system and the FlexRay drivers.

The implementation of the node application adds another limitation to the algorithm that is that the number of processes in a membership may not be larger than 32. This is

due to that a set is stored as a bit combination in the PowerPC in this implementation.

9.4 Future work

The future work needed to be done is to implement a more advanced real-time operating system so that our implementation of the membership algorithm can be used together with a more advanced application. As of today, the test platform consisting of a couple of nodes with a G2 board and a FlexRay board does not allow interrupt handling that well, which forces us to poll for events.

A good improvement for the future would be to extend the protocol so that it support several membership groups in the same system.

As Bergenhem is working on a new version of this protocol which is more fault tolerant it may be of interest to update this application to reflect that protocol.

Chapter 10

Acknowledgements

The most important person for the realization of this master's thesis is Carl Bergenhem, who has supplied us with a lot of articles and has helped us in many ways. So we wish to send many thanks to Carl for suggesting and assisting us with this interesting project.

We would also like to thank Mattias Bergström and Johan Högberg for their work on their thesis project which gave us much inspiration and has provided us with a good code base to base our work upon.

Many thanks to Rickard Svenningsson and Mansoor Chaudhry for their great introduction of how to work with the GAST boards, and how to use to ODEEP FlexRay Configurator.

Finally we would like to thank Mikael Rännar for being our assistant at the Department of Computing Science at Umeå Universitet.

References

[1] Cygwin. http://www.cygwin.com/ (visited 2007-06-29).

[2] Eclipse. http://www.eclipse.org/ (visited 2007-06-29).

[3] Flexray - the communication system for advanced automotive control applications. http://www.flexray.com/ (visited 2007-06-29).

[4] Gcc. http://gcc.gnu.org/ (visited 2007-06-29).

[5] Subclipse. http://subclipse.tigris.org/ (visited 2007-06-29).

[6] Subversion. http://subversion.tigris.org/ (visited 2007-06-29).

[7] Mfr4200 data sheet. Technical report, Freescale, P.O. Box 5405, Denver, Colorado 80217, 2006.

[8] M. Ayoubi, T. Demmeler, H. Leffler, and P. Köhn. X-by-wire functionality, performance and infrastructure. Presented at the Convergence Conf. 2004, Detroit, MI.

[9] C. Bergenhem and J. Karlsson. A configurable membership service for active safety technical report. 2007.

[10] M. Bergström and J. Högberg. Implementation of membership algoritms in gastcluster with flexray. Technical report, Chalmers University of Technology, 2007.

[11] G. Grünsteidl. Dezentrales redundanzmanagement in vertilten echtzeitsystemen. Technical report, Technical University of Vienna, 1993.

[12] R. Johansson and L.-Å. Johansson. G2 board, manual. Technical report, QRtech AB, 2007.

[13] R. Johansson, D. Lundgren, and S. Andersson. Rtcom boards, user manual. Technical report, QRtech AB, 2006.

[14] Nicolas Navet, Yeqiong Song, FranÇoise Simonot-Lion, and Cédric Wilwert. Trends in automotive communication systems. Technical report, Loria Laboratory, France, 2005.

[15] Ralf Schlatterbeck. Membership service: What it is and why you need one for a safety critical system. Technical report, TTTech Computer technik AG, 2002.

[16] R. Svenningsson and M. Chaudry. Flexray-based communication of diagnostic information in gast-environment. Technical report, Chalmers University of Technology, 2007.

[17] P. Souto V. Rosset and F. Vasques. A group membership protocol for communication systems with both static and dynamic scheduling. 2007.

[18] N. Vorkapic and J. Myhrman. Development of and open dependable electrical and electronics platform. Technical report, Chalmers University of Technology, 2006.

Appendix A

Glossary

ABS	Anti-lock Braking system.
BDM	Device used to update the software for the HCS12 controller.
CAN	Controller Area Network.
CEDES	Cost Efficient Dependable Electronic Systems.
Checker	Software to simplify the checking of data.
Consensus	To reach the same decition.
CVS	Concurrent Versions System.
Cygwin	A Linux like environment for Windows.
DEAD	Status of a node that does not work properly.
Eclipse	A IDE written in java.
ECU	Electronic Control Unit.
ELp	Electronics and Software.
ESP	Electronic Stability Program.
FD	Failure Detection Phase, first phase in the membership algorithm.
FlexRay	Time-triggered, with support for event-triggered, communication system for use in vehicles.
G1 and G2	Two ECU boards in the GAST project.
G2DBG	The debugger on the G2 board.
GAST	General Application Development Boards for Safety Critical Time-Triggered Systems.
GCC	GNU Compiler Collection.
GEE	GAST Eclipse Environment.
HCS12	Freescale microcontroller.
Hearbeat	A membership flag.
IDE	Integrated Development Environment.
ILF	Incomming Link Failure.
MC	Membership Communication Phase, second phase in the membership algorithm.
MD	Membership Detection Phase, third and last phase in the membership algorithm.
Membership algorithm	The algorithm that gives a correct view of the working nodes.

MFR4200	The FlexRay controller chip mounted on the RTComm from GAST.
MPC565	a CPU on the G2 board.
mT och μT	Macrotick and microtick.
NIT	Network Idle Time.
ODEEP FlexRay Configurator	A application to generate C-code to operate the FlexRay drivers.
OLF	Outgoing Link failure.
Opinion	A parameter in the membership algorithm that stores the current nodes opinion.
RTComm	Real Time Communication board for GAST.
RTOS	Real Time Operatin System.
S19	The file format that can be read of a CPU on a G2 board.
ScenarioParser	A application that translates the scenarios.
SerialBatchServer	A application that handles the connections to a GAST cluster.
SLOC	Source Lines Of Code.
SP	SP Technical Research Institute of Sweden.
Subclipse	A Subversion plugin for Eclipse.
SVN	Subversion.
TDMA	Time Division Multiple Access.
TTCAN	A version of CAN that is time-triggered.
TTP/C	Time-Triggered Protocol class C.
X-by-wire	Electronic control of something mechanical, e.g. brakes, steering.
Xml	Extensible Markup Language.

Table A.1: Glossary.

Appendix B

Membership Node Application user's guide

This is a node application that has been developed for a master's thesis project. The goal of the application has been to test how a membership algorithm works in a real environment consisting of a GAST cluster with several G2 cards coupled together with FlexRay cards as communication controller.

The algorithm which has been implemented has been developed by Carl Bergenhem, and is the third iteration of the algorithm. See the features list later for what this algorithm can handle.

This implementation is not intended to be used as a final product. Several limitations makes this unsuitable for real use. These limitations will be listed later.

B.1 Features

- Membership.
  - Dynamic reintegration.
  - Multiple processes per node.
  - Performs the membership communication/decision phase only when a change has been detected.
- Fault-injection down to a slot level.
- Non-blocking I/O.
- Can run both short scenarios and long real-time tests.
- Support for dual channel usage.
- Output available for both performance tests and correctness tests.
- Synchronization on the startup for all nodes.
- Several user programs for demo purpose.
- Nodes can handle loss of physical connection to the rest of the cluster.

B.2 Requirements

- GAST cluster consisting of G2 and FlexRay boards.
- The G2-DebuggerPPC-BETA software installed on the G2 board.
- Windows HyperTerminal or the SerialBatchServer software.
- All G2 boards in the GAST cluster connected to a PC.
- The GEE environment installed.
- The ODEEP FlexRay Configurator if any change to the FlexRay configuration is needed.

B.3 User manual

First the node needs to be compiled. In a terminal, use the command 'make' while standing in the project root to compile the program. This will generate a number of *.s19 files in the debug folder.

Then the *.s19 files needs to be uploaded to the node. If it isn't already uploaded, this is how you do:

Type 'ferase' in the debugger followed by 'q'. The memory where the program will be uploaded should now be erased. Type 'fload', the debugger will await the program. Now you should use the internal file sending command in the console you use to communicate with the G2 card. In HyperTerminal you use the command 'Send Text File...' and pick the *.s19 file to upload. Wait for it to finish. Then start the application by typing 'go 10000'.

The applications will need a number of parameters to start up. It is however possible to use the default values by just pressing enter on all the input fields to get a quick start.

Parameters v1.2 explained:

stop cycle How many cycles that will pass before the application quits.

number of nodes The number of nodes that will be connected to the cluster, a maximum of five.

active nodes at startup Takes a node set with all the nodes that will be synchronized.

number of processes Number of processes that will run on this node, a maximum of four.

process XX user program Select which user program will be run as process XX.

B.4 Fault-injection v1.0 explained

We suggest that you use the ScenarioParser application to create fault injection scenarios. The method used here is a bit tricky and requires the user to pay quite a lot of attention. However, here is how to use the fault injection on the node application manually.

When a fault is injected it is injected in a certain phase. For example, if a fault is injected in the failure detection phase in cycle 3, all the cycles after 3 that fault will

occur until another event is found. Important to realize is that this fault is not injected in any other phase, so for this to work in a realistic fashion you need to inject the same fault in the membership communication phase also.

Many of these inputs takes something we call a set. A set is actually just a binary number, but you can think of a binary number as a container that contains numbers. For example the number 5, that is written as the binary number 101, has the third and the first bit set. So this set contains the processes with number 1 and 3. The input sets are written in hexadecimal values.

A node set is a set that is built up of 5 bits, 1 bit for each node. A set of all the nodes would then be 1F in hexadecimal. A process set is a little more complex, since there is a maximum of 4 processes per nodes, there will be 20 processes available in this application (this setup is very strict since it is specified in the FlexRay configuration). The maximum value will then be 1048575 or FFFFF in hexadecimal value. The bits are distributed as following:

Node set:

```
       1 1 1 1 1
       ^ ^ ^ ^ ^
Node4 _| | | | |
  Node3 _| | | |
    Node2 _| | |
      Node1 _| |
        Node0 _|
```

Process set:

```
  Node4   Node3   Node2   Node1   Node0
|-------|-------|-------|-------|-------|
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                                 ^ ^ ^ ^
                       Process3 _| | | |
                         Process2 _| | |
                           Process1 _| |
                             Process0 _|
```

The following parameters will be asked for when specifying an event:

event Select where you want an event to be ejected.

cycle In which cycle the fault will be injected.

nodes/processes A set containing the nodes/processes that will be alive at this cycle in the specified phase until another event is injected.

Type 'q' as event to finish.

The application will then start up, and depending on which user programs you selected you will get different output. If you selected cycle 0 as the stop cycle the program will continue to run until the power is cut, or a user program terminates the OS.

How the output data works will not be described here, you will have to figure that out yourself.

B.5 User programs

Sample prog Simple program that prints the alphabet one character at a time, every second. For monitoring purpose the current membership opinion is also printed out each second. A node that gets an error for some reason and gets OK again will integrate with the other nodes and print the alphabet at the same point as all the other processes in the membership.

Monitor prog This is a utility user program that can be handy to use while doing a long test run. This program will print the membership on any change. It also provides the functionality to quit the node application by typing 'q'.

Jage prog Jage is a kids game where one kid hunt the other kids. If the hunter catches another kid, that kid will be the new hunter. This program is an example of a little more advanced program you can create using membership. All processes running this program and is part of the membership in the cluster corresponds to a player in the Jage game.

Only sample prog and Jage prog sends a heartbeat every cycle, so it is recommended to only run monitor prog as secondary a node process.

B.6 FlexRay setup files

The FlexRay setup files are saved in XML format and is the format supported by the ODEEP FlexRay Configurator. 'cluster_version4.XML' is the final setup which is used in this implementation.

B.7 Limitations

- The code right now supports up to 5 nodes with 4 processes on each node. This is quite easily updated by changing the FlexRay configuration and the cluster initialization process.
- Due to how the set work it is not possible to have more than 32 processes included in the membership.
- The RTOS can not control the user programs. If any user program takes too long to execute the node will be excluded from the membership.

Appendix C

Checker user's guide

The checker is a small program which put together the result from several output files from a test run to a file which is easier to interpret. The checker has different levels of how much output it gives to the summary file.

C.1 Requirements

Java 1.6.

C.2 User manual

The programs is located in the jar files under the dist folder.

To run the program use the command 'java -jar dist/checker.jar'. The first parameter is obligatory, it is what level of output wanted in the summary file. 0 = as little output as possible, only writes out if there are any errors or if gold differs. 1 = all as in 0 and also a summary of all nodes membership vector every cycle the test ran. 2 = all as in 1 and also writes the gold even if it doesn't differs. The second parameter is also obligatory, it is the name of the file where the summary shall be written down. The following parameters are the names of the files that came from the test run. For every name the program tries to find an scenario file that ends with '.input' instead of '.txt'. This is just used if they exist and prints the scenario in the summary to make it easier to know what was run.

Example: `java -jar dist/checker.jar 2 summary.txt Node0.txt Node1.txt Node2.txt Node3.txt Node4.txt`

Appendix D

ScenarioParser user's guide

The scenario parser put together an input file for every node. The input file consist of the input parameters from one scenario file, with the same format as the input file to the simulator. The input parameters are for example, for how long the test shall carry on, number of processes, if and what type of fault and so on...

D.1 Requirements

Java 1.6.

D.2 User manual

The programs is located in the jar files under the dist folder.

To run the program use the command 'java -jar dist/parser.jar'. The first parameter is obligatory, it is the name of the scenario file. The second parameter is also obligatory, it is the number of maximum processes per node.

Example: `java -jar dist/parser.jar scenariofile.txt 4`

D.3 File format

The file format that the ScenarioParser use is shown below.

```
// is a comment.

// The following parameters are used:
// nbrnodes = cluster size
// simmegacycles = number of cycles
// idlebeforejoin = backoff time
_simparameters
description short duration ILFS 3/4 way in MH
nbrnodes 5
simmegacycles 10
sslotspercycle 10
dslotspercycle 7
```

```
networkidle 1
cyclespermegacycle 1
ILFhearown NO
simmode  -12
idlebeforejoin 0
membershipversion 3
// procXY s1 s2 c = process X at node s1 is sending at static slot s2.
_snodetraffic
//      s s c
proc00  0 0 1
proc01  1 1 1
proc02  2 2 1
proc03  3 3 1
proc04  4 4 1
proc10  0 5 1
proc11  1 6 1
proc12  2 7 1
proc13  3 8 1
proc14  4 9 1
end
// mopx s c mid = membership opinion from node s is sent in dynamic
// slot with id mid
_dnodetraffic
//    s c mid
mopa  0 1 0
mop1  1 1 1
mop2  2 1 2
mop3  3 1 3
mop4  4 1 4
bla1  0 1 6
bla2  2 1 7
bla3  3 1 8
bla4  4 1 9
end
// nodeX src megacycle cycle slot action = action at node src starts
//    in megacycle at slot. ((src == X), actions can be OK, OFF)
// procXY megacycle cycle slot action = action in procXY starts in
//    megacycle at slot.      (actions can be OK, ILF, OLF, SIL)
// goldX megacycle cycle slot {Member||NotMember}*nbrOfProcess = golden
//    vector is the correct view.
_faultevents
node1 1  2 1 13 ILFS
node1 1  2 1 16 OK
gold1  5 1 0 M M M M M M M M M M
gold2  4 1 0 M M M M M M M M M M
end
endfile
```

Appendix E

SerialBatchServer user's guide

For the master's thesis project there was a need for an application to upload user programs to the node cluster in a convenient and fast way. The manual way of doing it via the windows application HyperTerminal took too long and was very awkward.

This application started out as a very simple program for uploading files to multiple serial connection. But the application evolved and eventually became a very convenient program for communicating and working with the GAST G2 card. Although this application is developed with the GAST G2 card in mind, this application is very general and should work with any device connected to the serial port.

The application works in a server-client philosophy. This also allows other computers to work with the cluster even though they are not physically connected to the cluster.

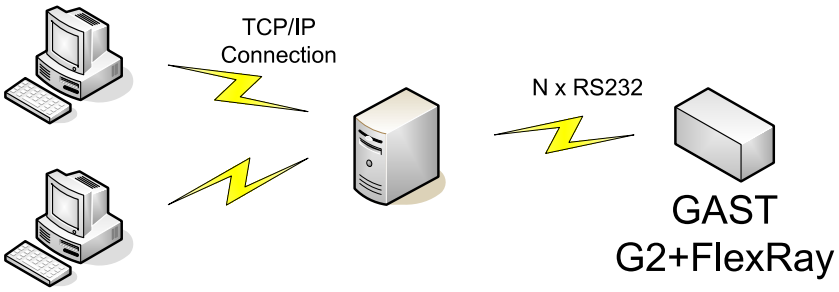

Figure E.1: Diagram showing how the node cluster is connected to a PC via a server.

E.1 Features

- Direct communication with the cluster via remote terminals.
- It is possible to connect several remote terminals to every node.
- Node settings are specified with the XML file format specified in Appendix E.4.
- In the node settings file it is possible write a macro to the node.
- Progress bar when uploading files.

– Queues job if the node cluster is busy by another user.

– Automatic power switch to the cluster if the correct controller switch is connected to the serial port.

E.2 Requirements

– Java runtime environment 1.6.

– RXTX drivers for serial port communication (http://www.rxtx.org).

– Enough serial ports to connect to the cluster.

E.3 User manual

The programs is located in the jar files under the dist folder. First start up the server, then connect the clients to that server. There is no need to manage the server, once it is started it will take care of itself.

Server startup: `java -jar dist/server.jar <parameters>`

Parameters available are:

```
Server port: [-p|--portnum] <port number>
Alive test: [-a|--alive] <alive command> <expected number of bytes>
Serial ports: [-z|--ports] {<port name> <port output number>,
              <port name> <port output number>...}
Port parameters: [-r|--params] <baud rate> <data bits> <stop bits>
              <parity>
Power controller port: [-c|--powerport] <port name>
Abort on client disconnect: [-b|-abort]
```

Example: `java -jar dist/server.jar -p 1234 -z COM3 1235 COM4 1236 -a "help" 369 -c COM2 -b`

Note: The port output number for the serial ports are the port on which it is possible to connect a remote terminal.

Client startup: `java -jar dist/client.jar <server address> <server port> <XML file>`

Server address: The address on which the server is located, use localhost if it is on the same computer. Server port: The port on which the server was told to listen for connections. XML file: The node settings in the XML file format.

Example: `java -jar dist/client.jar localhost 1234 example/example.xml`

Remote Terminal startup: `java -jar dist/terminal.jar <server address> <server port>`

Server address: The address on which the server is located, use localhost if it is on the same computer. Server port: The port on which the server was told to listen for connections for the node you want to talk to.

Example: `java -jar dist/terminal.jar localhost 1235`

E.4 XML file format

The XML file looks like following:

```
<?xml version="1.0"?>
<SerialConnections>
   <Port name="COM1" outFile="output1.txt">
      <inplace postDelay="1000" preDelay="0" bytesExpected="369"
         type="command">help</inplace>
      <inplace postDelay="1000" preDelay="0"
         type="command">ferase</inplace>
      <inplace postDelay="1000" preDelay="0"
         type="command">q</inplace>
      <inplace postDelay="1000" preDelay="0"
         type="command">fload</inplace>
      <file postDelay="0" preDelay="0"
         type="data">example\file.s19</file>
      <inplace postDelay="0" preDelay="5000"
         type="command">go 10000</inplace>
   </Port>
   .
   .
   .
</SerialConnections>
```

For each port you want to connect to you add another <Port> block. In the Port tag name declares which port you want to connect to, and outFile where the output is stored.

For each command you want to specify in the XML you add a tag called inplace. The string entered between the start and the end tag will be sent to the serial port.

If you want to specify the commands in a separate file you use the <file> tag, and the name of the file is entered between the start and the end tag.

The preDelay and the postDelay specifies how long the server will wait before, respectively after the command has been sent. It is specified in milliseconds. If used together with a file you should note that this will delay after each line is sent, or each chunk of data is sent (depending on whether you specified it as data, command or hex). If you want a little delay after a file is sent you should specify a preDelay on the command following the file.

There are three types for the string, those are *command*, *data* and *hex*.

If it is sent as a *command* a newline will be sent together with the string. If it is loaded from a file a full line will be read and then sent together with a newline char.

If it is sent as *data* it will be sent as it is typed. If it is loaded from a file the file will be sent in chunks of x number of bytes, exactly as it looks.

If it is marked as *hex* the string will be interpreted and sent as integers. Each integer is entered by its ASCII sign, and is separated by either a white space or a newline. If it is separated by a newline you have the chance to specify a delay between each of the values sent.

bytesExpected was previously used to test the connections. But since version 8 of this application it is no longer in use.

Appendix F

TestPlatform user's guide

The TestPlatform contains a number of batch scripts that tie all the applications (used for testing the membership node application) together.

Using these scripts makes testing scenario files much more efficient than using the application on their own.

F.1 Features

- Possible to compile, upload and test with only one command line.
- Can test single files, and also whole folders with scenario files.

F.2 Requirements

- The SerialBatchServer package available.
- The Checker application available.
- The ScenarioParser application available.
- The GEE environment installed.

F.3 Installation

1. The file path of the environment should be 'C:\MembershipData'. If it is in another file path the XML files need to be updated to reflect that.
2. Modify the header of 'make.bat' so that it points to the correct file paths.
3. Modify the header of 'run.bat' so that it points to the correct file paths.
4. Modify the header of 'server.bat' so that it points to the correct file paths, this file also needs to be updated so the correct serial ports are loaded at server startup.
5. Modify the header of 'terminal.bat' so that it points to the correct file paths, this needs to be changed so that it connects to the ports that was specified in the 'server.bat' script.

F.4 Scripts

None of the scripts is really needed, they are all available to simplify the work process.

make.bat: Compiles the program referred to in the file, and copies the binaries created to the 'input' folder.

run.bat: Run a scenario file on the cluster, a server needs to be up and running for this to work.

server.bat: Starts a server with the default parameters.

terminal.bat: If the server is started with the default parameters you can connect to a node with this program.

F.5 User manual

All commands are typed in the command prompt, start the command prompt by pressing Run in the Start menu, and typing cmd in the run window.

Start up a server:

1. Open a new command prompt.
2. Type 'cd /D C:\MembershipData'.
3. Start the server by typing 'server.bat'.

Run a scenario:

1. Open a new command prompt.
2. Type 'cd /D C:\MembershipData'.
3. Type 'make.bat' to compile the membership node application.
4. To run a single scenario file, in the command prompt in C:\MembershipData\ type: 'run file <scenariofile> <XML file>'
5. To run a whole scenario dir, in the command prompt in C:\MembershipData\ type: 'run dir <scenariodir> <XML file>'
6. It is possible to repeat the last run scenario by typing: 'run run <XML file>'
7. All output data will be placed in 'C:\MembershipData\output\ <scenario dir|file>'.

There are two XML files available as default in this package.

- 'go.xml' is used to just run the node application.
- 'upload_and_run.xml' is used to first upload the node application and then run it.

Connect to a node via the server:

1. Open a new command prompt.
2. Type 'cd /D C:\ MembershipData'.
3. Type 'server.bat #' where # is the number of the node you want to connect to. # is the actual number written on the node cluster, but this is obviously very dependent on the setup you have done before.

F.6 Tips

– Type 'make run [file|dir] <scenario file|dir> upload_and_run.xml' to compile, upload and run with a single command.

Appendix G

Cluster automated power switch

The power switch is used to control the power to the cluster via the SerialBatchServer. The relay is connected between the power supply and the cluster to break the 12 VDC and is controlled by a connection to a PC. The PC is connected via the RTS signal on a RS232 port. Pin 7 is the RTS signal and it needs to be amplified to maneuverate the relay. This is done with a transistor.

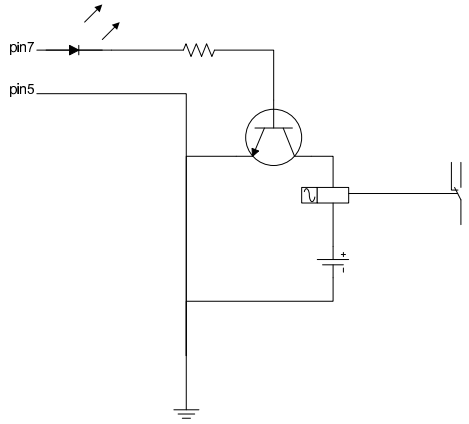

Figure G.1: Schema over the COM-port component used together with the GAST cluster to serve as an automated power switch.

Appendix H

Algorithms

H.1 The decision function

The decision function used in all versions of the membership protocol.

```
df_maj(vector votes, int u)
{
   if (u/2 or more opinions are member)
      then Mp = member
   else if (u/2 or more opinions are not member)
      then Mp = not member
   else
      then Mp = undefined
   return Mp
}
```

H.2 The basic protocol with reintegration

The following steps are performed by the protocol in each node.

H.2.1 States

- The *membership* vector contains the status of all processes, initially all processes have status *member*.
- The *opinion* vector contains the status of messages received during FD. It is the candidate membership vector.
- u is the upper bound of the number of expected opinions during the MC phase. It is used when deciding the new status of membership. Initially set to the total number of process, N.

H.2.2 During the FD phase

1. If I am active then broadcast heartbeat messages from my processes

2. Else if I host a process that wishes to join the group
   - Then set relevant processes to member in my *opinion* and set my node to active
     - If my node was reactivated set my u to N and set all processes to *member*
   - Broadcast join-request message for joining process
3. Observe the heartbeat messages from processes that I consider to be members
   - Take note in my opinion vector
4. On reception of join-request message
   - Set joining process to *member*

H.2.3 During the MC phase

1. Empty *opinionmatrix*
2. Broadcast *opinionid* and my u to all nodes
3. If an incoming opinion and u belongs to an active node, take note in *opinionmatrix* and note u
4. If it belongs to a reactivating node, take note in *joinopinionmatrix* and note u

H.2.4 During the MD phase

1. Get decision function result concerning each process. Perform *dfresult = df_maj(opinioncol, u)*, column by column of opinion matrix. and with minimum of all received u
2. If for any process *dfresult*
   (a) is *undefined*
      - Set my node to not active
   (b) differs from my opinion and I am active
      - Set my node to not active
   (c) is not a subset of my opinion and I am joining
      - Set my node to not active
   (d) is not *member* for a process that I host
      - Set my opinion about the process to *not member*
   (e) if I do not host any member processes
      - Set my node to not active
3. If any active node sent an opinion that differs from *dfresult* or if any reactivating node sent an opinion that is not a superset of *dfresult* then set all processes hosted by that node to not active
4. Set my u to the number of active nodes in my *opinion*
5. If an active nodes did not broadcast an *opinion* during MC

  - Set to *not member* all processes hosted on that node

6. If I do not host any member processes then set my node to not active
7. Set my new *membership* = my *opinion*

H.3 The membership protocol with dynamic operation

The following steps are performed by the protocol in each node:

H.3.1 States

- The *membership* vector contains the status of all processes, initially all processes are *member*.
- The *opinion* vector contains the status of messages received during FD. It is the candidate membership vector.
- u is the upper bound of the number of expected opinions during the MC phase. It is used when deciding the new status of membership. Initially set to the total number of processes, N.
- *genid* is the generation identification of my membership group. Initially set to 0.
- *M-request* indicates that membership handling (MC and MD-phases) shall be performed. Initially set to false.

H.3.2 During the FD phase

1. If I am active then broadcast heartbeat messages from my processes, with *M-request* as determined in the previous MD-phase
2. Else if I host a process that wishes to join the group
   - Set relevant processes to member in my *opinion* and set my node to active
     - Set *M-request*
     - If my node was reactivated set my u to N; and set all processes to member; my *genid* to 0
   - Broadcast join-request message for joining process
3. Observe the heartbeat messages from processes that I consider to be members
   - Take note in my opinion vector
4. On reception of join-request message
   - Set joining process to *member*
5. If (I received a heartbeat message with *M-request*) or (modified my *opinion* (to *not member*) in step 3 or (to *member*) in step 4)
   - Set *M-request*

H.3.3 During the MC phase

– If M-request is set

1. Empty *opinionmatrix*
2. Broadcast *opinionid* and my *u* and *generationid* to all nodes
3. If an incoming *opinion* and *u* belongs to an active node, take note in *opinionmatrix*, note *u* and *generationid*
4. If it belongs to a reactivating node, take note in *joinopinionmatrix* and note *u* and *generationid*

H.3.4 During the MD phase

– If M-request is set

1. If I am joining then set my genid to the maximum of all received genids else if my genid differs from the maximum of all received genid then cease membership operation
2. Get decision function result concerning each process. Perform *dfresult = dfmaj(opinioncol, u)*, column by column of opinion matrix. and with minimum of all received u. Exclude opinions received from nodes that have a differing genid
3. If for any process *dfresult*
   (a) is *undefined*
      • Then set my node to not active
   (b) differs from my opinion and I am active
      • Then set my node to not active
   (c) is not a subset of my opinion and I am joining
      • Then set my node to not active
   (d) is *not member* for a process that I host
      • Set my opinion about the process to *not member*
      • If I am no longer active, then cease membership operation
4. If any active node sent an opinion that differs from dfresult or if any reactivating node sent an opinion that is not a superset of dfresult
   • Then set all processes hosted by that node to not member
5. Set my *u* to the number of active nodes in my *opinion*
6. If an active nodes did not broadcast an *opinion* during MC
   • Then set to *not member* all processes hosted on that node
7. If I set any process to *not member* in step 6
   • Then set *M-request*
   • Else reset *M-request*
8. If I do not host any member processes then set my node to not active
9. Set my new *membership* = my *opinion*. Increment my genid